Processes to Produce Secure Software

Towards more Secure Software

Volume I

Software Process Subgroup of the Task Force on Security across the Software Development Lifecycle

National Cyber Security Summit

March 2004

Edited by Samuel T. Redwine, Jr. and Noopur Davis

Processes to Produce Secure Software

Towards more Secure Software

Volume I

Software Process Subgroup of the Task Force on Security across the Software Development Lifecycle

National Cyber Security Summit

March 2004

Edited by Samuel T. Redwine, Jr. and Noopur Davis

Foreword

The Software Process Subgroup within the Task Force on Security across the Software Development Lifecycle of the National Cyber Security Summit – Co-Chaired by Sam Redwine (JMU), Geoff Shively (PivX), and Gerlinde Zibulski (SAP) – produced this report. The Task Force and its Subgroups were established December 2-3 in Santa Clara, California at the National Cyber Security Summit sponsored by the Department of Homeland Security (DHS) and several industry groups – Business Software Alliance, Information Technology Association of America, Tech Net, and US Chamber of Commerce. The three-month effort to produce this report, spanning December 2003 through February 2004, is part of the DHS-private sector partnership. The Subgroup's life should extend beyond the production of this report, but its specific activities may vary.

Editors:

Samuel T. Redwine, Jr.
Noopur Davis

Authors:

Noopur Davis
Michael Howard
Watts Humphrey
Gary McGraw
Sam Redwine
Gerlinde Zibulski
Caroline Graettinger

Additional contributors included Roderick Chapman and Anthony Hall (Praxis Critical Systems), Joe Jarzombek Office of the Assistant Secretary of Defense (Networks and Information Integration), Richard C. Linger (Software Engineering Institute), Peter Neumann (SRI), and Stacy J. Prowell (University of Tennessee). Papers prepared for the Subgroup appear in VolumeII.

The editors want to thank task force members, particularly their co-authors, for the many review comments received and the helpful issue discussions that occurred. In part, Sam Redwine's work was supported by Virginia's Commonwealth Technology Research Fund and the NIST's Critical Infrastructure Protection Project through the Institute for Infrastructure and Information Assurance (3IA) at James Madison University.

Special thanks go to Ron Moritiz (Computer Associates) and Philip Reitinger (Microsoft) for participating in early teleconferences, Cindy Gagliano (Booz Allen Hamilton) and Plonk Audrey (US CERT) for their administrative help, Jim Kohlenberger (Business Software Alliance) for his leadership and administrative help, and to Caroline Graettinger (Software Engineering Institute) for the section on Organizational Change.

Software Process Subgroup membership:

Leslie Beach – SRA International
Noopur Davis – Software Engineering Institute
Kenneth Dill – PivX Solutions

Dana Foat – OSD(NII) DIAP Defense-wide Information Assurance Program
Richard George – National Security Agency
Kwang Ho Kim – AlphaInsight Corporation
Michael Howard – Microsoft
John Hudepohl – Nortel Networks
Watts Humphrey – Software Engineering Institute
Joe Jarzombek – Office of the Assistant Secretary of Defense (Networks and Information Integration)
Lalita J. Jagadeesan – Lucent Technologies
James Lewis – Center for Strategic and International Studies
Steve Lipner – Microsoft
Paul Lloyd – HP
Gary McGraw – Cigital
Sam Redwine – James Madison University
Geoff Shively – PivX Solutions
Srinivasa Venkataraman – Software executive
Peggy Weigle – Sanctum
Ulrich Werner – SAP
Gerlinde Zibulski – SAP

Any corrections or comments regarding this report should be sent to Sam Redwine – redwinst@jmu.edu.

Executive Summary

The Software Process Subgroup addressed the process issues raised by the Security-across-the-Software-Development-Lifecycle Task Force of the National Cyber Security Summit. This subgroup report defines a path for software producers to follow in producing secure software and it includes recommendations to software producing organizations, educators, and the Department of Homeland Security (DHS) on how to motivate and aid software producers in following these recommendations.

The Problem

Security is now a serious problem and, if present trends continue, the problem will be much worse in the future. While there are many reasons for security problems, a primary cause is that much of the software supporting the US cyber infrastructure cannot withstand security attacks. These attacks exploit vulnerabilities in software systems.

Software security vulnerabilities are caused by defective specification, design, and implementation. Unfortunately, common development practices leave software with many vulnerabilities. To have a secure US cyber infrastructure, the supporting software must contain few, if any, vulnerabilities. This requires that software be built to sound security requirements and have few if any specification, design, or code defects.

Software specification, design, and code defects are unknowingly injected by software developers and, to produce software with few defects, common development practices must change to processes that produce software with very few defects. This requires that developers use methods that consistently produce secure software, which in turn requires development organizations to acquire a high level of security expertise, identify and adopt processes for producing low-defect, secure software, and consistently use this security expertise and these processes when they produce, enhance, maintain, and rework the software that supports the US cyber infrastructure.

Current Status

No processes or practices have currently been shown to consistently produce secure software. However, some available development practices are capable of substantially improving the security of software systems including having exceptionally low defect rates. Since introducing these methods requires significant training and discipline, they will not be widely adopted without strong motivation from sources such as corporate leaders, customers, or regulation.

Required Actions

This report describes the actions required to address the current situation. The principal actions are to broaden use of the currently most promising available practices for developing low-defect, secure software, to produce definitive studies that compare the relative effectiveness of available security practices, and to work within the software industry to achieve widespread use of the most effective security practices. A comprehensive program to validate that candidate software processes consistently produce secure software is also needed.

Recommendations

The principal recommendations in this report are in three categories:

Principal Short-term Recommendations

- Adopt software development processes that can measurably reduce software specification, design, and implementation defects.
- Producers should adopt practices for producing secure software
- Determine the effectiveness of available practices in measurably reducing software security vulnerabilities, and adopt the ones that work.
- The Department of Homeland Security should support USCERT, IT-ISAC, or other entities to work with software producers to determine the effectiveness of practices that reduce software security vulnerabilities.

Principal Mid-term Recommendations

- Establish a security verification and validation program to evaluate candidate software processes and practices for effectiveness in producing secure software.
- Industry and the DHS establish measurable annual security goals for the principal components of the US cyber infrastructure and track progress.

Principal Long-Term Recommendations

- Certify those processes demonstrated to be effective for producing secure software.
- Broaden the research into and the teaching of secure software processes and practices.

Table of Contents

Introduction

Scope and Purpose

Today, security problems involving computers and software are frequent, widespread, and serious. The number and variety of attacks by persons and malicious software from outside organizations, particularly via the Internet, are increasing rapidly, and the amount and consequences of insider attacks remains serious.

This report concentrates on the processes and practices associated with producing secure software. It mentions only in passing physical, operational, communication, hardware, and personnel security. These are important topics but outside the scope of this report. Concentrating on software, however, still covers the bulk of the security vulnerabilities being exploited today – the ones in software.

Software security issues have long been studied, and, while open questions remain, considerable bodies of research and practices to address them exist. This report outlines known practices, recommended actions, and research needs, and is intended for the use of software producers, the US Department of Homeland Security, and others interested in improving the processes used to build software with security requirements. While limited by the three months available for its production and the best knowledge of those involved, this report provides substantial useful – albeit not exhaustive or all-knowing – information and guidance for producers of software and those interested in improving the current situation.

Software Security Goals and Properties

The primary goals of software security are the preservation of the confidentiality, integrity, and availability (CIA) of the information assets and resources that the software creates, stores, processes, or transmits including the executing programs themselves. Preserving confidentiality is about preventing unauthorized disclosure; preserving integrity is about preventing unauthorized alteration; and preserving availability is about preventing unauthorized destruction or denial of access or service. The property of non-repudiation, ensuring the inability to deny the ownership of prior actions, can be of special interest.

Security is not just a question of security functionality; the properties desired must be shown to hold wherever required throughout the secure system. Because security properties are systems properties, security is an omnipresent issue throughout the software lifecycle. [McGraw 2003]

In addition to the preservation of these properties within its digital domain by a software system, other systems, organizational, or societal security goals can be contributed to by software including:

- Establishing the real-world authenticity of users and data
- Establishing accountability of users
- Permitting usability so as to gain users' acceptance of security features,
- Providing the abilities to deter and mislead attackers, detect attacks when they happen, notify when they occur, continue service, confine their damage, rapidly

recover from them, easily repair software to prevent future attacks, and investigate the attackers

As well as this ability to tolerate and recover from effects of attacks, the ability of a system to defend in depth with multiple layers of defense is also desirable. Deciding the extent of security-oriented development effort and functionality is a risk management decision. Whatever one decides, the required security properties need to be explicitly defined. Neither in the physical world nor for software can security be absolutely guaranteed. Thus, when this report speaks of "secure software" the true meaning is "highly secure software realizing – with justifiably high confidence but not guaranteeing absolutely – a substantial set of explicit security properties and functionality including all those required for its intended usage."

In the remainder of this report we use the following security terminology – hopefully already familiar to many readers. Threatening entities or agents may possess or be postulated to possess certain capabilities and intentions creating threats. Threats utilize vulnerabilities in the system to perform their attacks. Adversaries use specific kinds of attacks or "exploits" to take advantage of particular vulnerabilities in the system. Systems may have countermeasures to reduce certain vulnerabilities. See Figure 1: Security Concepts and Relationships (Source: Common Criteria) for relationships among these terms.

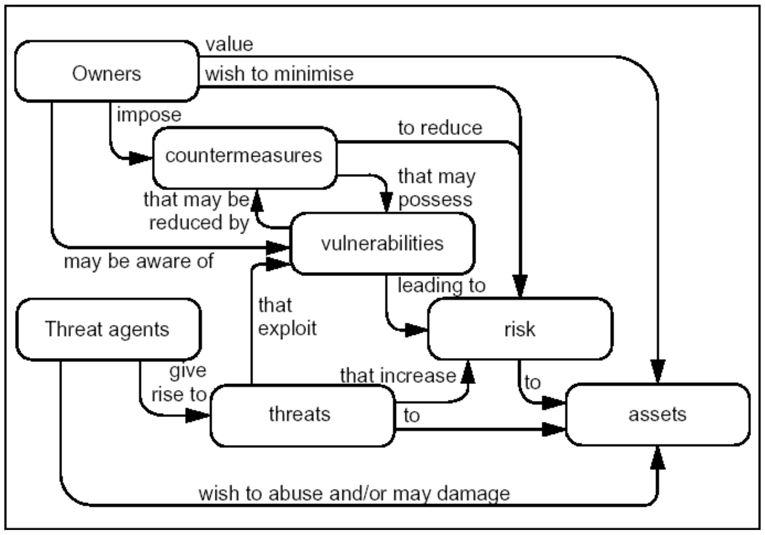


Figure 1: Security Concepts and Relationships (Source: Common Criteria)

As an example, certain classes of vulnerabilities such as buffer overflows have proven common in current software – despite the existence of known ways to avoid putting them into software.

This is just one example of the widespread failure to utilize known practices – later a number of these are identified in the Practices section of this report. Encouraging and facilitating increased use of these practices is a central theme of this report.

Software Process

The first books enumerating steps for a process to produce software appeared in the early 1960's – if not earlier. Software process has been an active area of work in industry, research, and government ever since – within this has been significant work on processes for high-dependability systems. Today, a plethora of books contain mainly general purpose practices and processes. These range from lightweight processes placing few requirements on developers to heavyweight ones that provide a high level of guidance, discipline, and support. [Boehm] Generally and not surprisingly, success in producing high-dependability systems aimed at safety or security has been greater with software processes closer to the heavyweight end of the spectrum and performed by highly skilled people.

To reliably produce secure software, one needs three things:

1. An outstanding software process performed by skilled people
2. A sound mastery of the relevant security expertise, practices, and technology
3. The expert management required to ensure the resources, organization, motivation, and discipline for success

Achieving these three things will require considerable effort by organizations that already have simply a good software engineering process and even more from the bulk of organizations that fall short of having even this. These processes and the required skill and expertise to carry them out are the central issue in this report. Improving software engineering practices and processes can not only lead to secure software but to software released with few defects, with reasonable costs of development, with lower maintenance costs, and with an enhanced reputation for the product.

Organization of Report

In addition to its Executive Summary, Foreword, and this Introduction, this report contains sections on:

- **The Problems** involved in producing secure software
- **Requirements** for processes and practices to produce secure software
- **Practices** for producing more secure software
- **Organizational Changes** needed for introduction, use, and improvement of processes and practices
- **Qualifications:** Verification, validation, and approval of processes, practices, people, and products
- **Recommendations**

The problem section covers security and the current situation, and views of the problem from industry, risk management, process, technical, product, and organizational perspectives. The requirements section enumerates a set of required properties for a process (or practice) to produce secure software in which one can have justifiable confidence. Related issues are addressed in the Qualification section.

The bulk of this report is dedicated to describing current practices. While brief and only covering a subset of the candidates for leading practices, these should prove to include items that many can take immediate action on.

A section on the introduction, use, and improvement of processes and practices is included to introduce organizations wishing to improve to the many issues involved in organizational change and the body of knowledge about how to address them.

This report ends with sections on recommendations to industry, government, and academia covering the near, mid, and long terms; and final conclusions.

The Problem

Current Software Security Problem is Serious

Intrusion and malicious software cost US industry and government ten plus billion dollars per year and potential attacks on critical infrastructure remain a serious concern. New automatic attack triggers require no human action to deliver destructive payloads. Security incidents reported to the CERT Coordination Center rose 2,099 percent from 1998 through 2002 – an average annual compounded rate of 116 percent. During 2003, the total was *137,529 incidents up from 82,094 in 2002.* An incident may involve one to hundreds (or even thousands) of sites and ongoing activity for long periods. These incidents resulted from vulnerabilities. *Figure 2* shows the yearly number of vulnerabilities reported to CERT CC. These can impact the critical infrastructure of the US as well as its commerce and security.

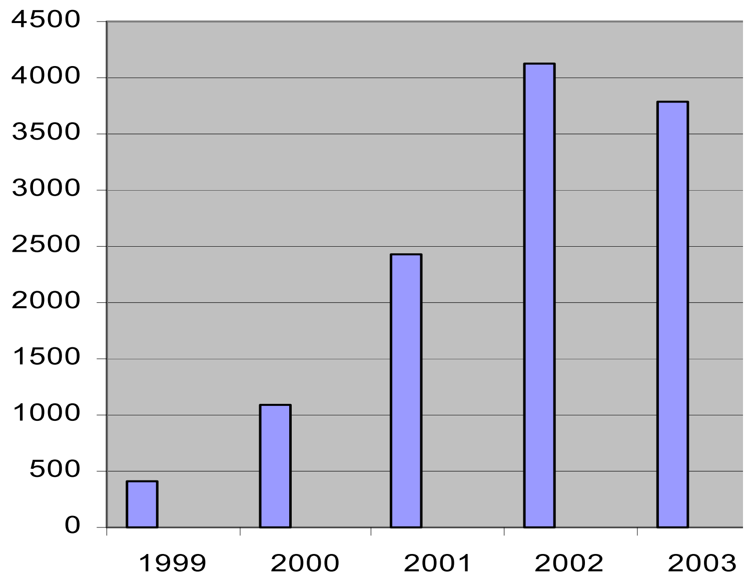


Figure 2: Vulnerabilities Reported to CERT CC

The substantial costs of a vulnerability to producers result from a number of activities – initial testing, patching, remediation testing, and distribution, as well as negative impact on reputation. Thus, producers can suffer serious consequences.

The losses of confidentiality resulting in identity theft or stolen credit numbers are frequently reported. Substantial fraud losses through unauthorized changes or transactions violating integrity are occurring. Denial of service attacks have occurred against major Internet e-commerce sites. For most software producers, however, combating these and reducing their and society's costs by producing secure software faces severe problems. The problems in producing software with the proper confidentiality, integrity, and availability properties compound the existing problem of producing quality software – already a serious one. Another way of saying this is that the quality of software (as measured by vulnerabilities) is frequently not adequate to meet the demands of the operating environment.

Problem of Producing Secure Software is Complex

For many reasons, producing secure software is complex. Connection to the Internet and provisions for user extensibility introduce elements uncontrolled by the original developer. The software product and its code are often already large and complex themselves. Code has multiple roles as both a real-world actor and an engineering representation.

Software's production is a complex process involving many activities and specialties. The types of activities range from requirements through maintenance plus management and support tasks – each produces its own products. Dozens of specialties exist. For example, just within design we have database design, network design, human interface design, and security engineering design as well as the mainstream software architectural and detailed design specialties. Other specialties such as business process, performance, safety, and reliability engineering may also contribute. Thus, dozens of different kinds of specialties and products are involved in a large development effort ranging from the highly technical to management status reviews.

Computer science and security are deep subjects. As just one security-related example, when two or more software components each possessing a security property are combined, the resulting combination may not exhibit the property. Furthermore, the analysis to establish if this is true can be quite subtle. And, in order to be done well, software engineering and security engineering require serious engineering endeavors.

Producing quality software requires personnel with substantial education, training, and experience. The project management involved in a substantial software project is also significant with management issues causing more failed projects than technical ones.

The problems in correctly implementing the required security-related properties and functionality encompass all those normally involved in producing defect free software on time and within budget – plus issues involving security. Without analysis, any specification, design, or code defect in the software may be potentially exploitable – thus, the extreme need for few or no defects. Furthermore, verification needs to cover guarding against intelligent adversaries – not just random adversity.

Justifiable confidence in the product can be created by design and construction in a manner that is shown – possibly partially mathematically – to preserve the required properties and provide security with an extremely low defect rate. To be able to do this, the system design and preferably code need to be analyzable, which, given the state of the art, restricts the structures that may be used. In addition, one can verify the absence of known security defect types, [Whittaker] [Hogland] but, while quite useful, ultimately this is not a substitute for proper design and construction.

The software development organization and environment must itself be secure. While extensive, automated tool support has limitations. Finally, the software is sure to evolve and change.

Thus, the problem of producing software with the proper confidentiality, integrity, availability, and non-repudiation properties compounds the existing problem of producing quality software, which is already a serious one. No easy answer exists to the problem of building quality secure software.

Problem of Formally Defining Secure Software Is Complex

The problem of establishing the security properties and functionality a system should have, both today and in the future, is difficult. Security functionality can be extensive;

Part 2 of the Common Criteria document takes 155 pages to enumerate possible security-related functionality and has an equal number of pages of details and notes. [Common Criteria Part 2] Security requirements are not just a question of listing security functionality; the properties required must be shown to hold wherever required throughout the secure system. Contrary to what most users and even many developers assume, security functionality does not necessarily provide genuine security; security is a systems property emerging from the totality of system behavior.

The first necessity for secure software is specifications that define secure behavior exhibiting the security properties required. The specifications must define functionality and be free of vulnerabilities that can be exploited by intruders. The second necessity for secure software is correct implementation meeting specifications. Software is correct if it exhibits only the behavior defined by its specification – not, as today is often the case, exploitable behavior not specified, or even known to its developers and testers.

What kind of security and privacy are required, and what are its costs and risks are hard questions. Technical judgment is not enough; management and marketing judgments are required particularly when customers have limited interest in security or paying for it – or possibly do not ask for it because they assume it is already included.

Why are Existing Approaches Not in Wide Use?

Largely driven by needs for high reliability and safety, approaches to building high-dependability software systems exist today. As we will see in the section on Practices, mating these with security concerns has already resulted in processes used on real projects to produce secure software.

Doing what is required and carrying it to the extremes necessary is difficult on all levels – individual, team, project, and organization – and includes others if out sourcing or acquisition are involved. While, for current successful users, evidence exists that once achieved such processes may take less time and effort, reaching there takes substantial amounts of both.

This cost and required organizational courage and strain is one reason few organizations currently use these practices for producing secure software – we address this problem in the Organizational Change section – but other reasons exist. The first is that many organizations do not recognize such approaches exist. This document is a step toward solving this as are other current publications. [ACM] [Anderson] [IEEE] Another reason is the belief that security is not required for product success. This comes from the existence of other goals that are viewed as more important such as user convenience, additional functionality, lower cost, and speedy time to market plus evidence that these are what have sold products in the past.

On the other hand, legal demands for secure software are increasing; one example is HIPAA in the health field,[1] and, if software security problems persist, the spectre exists of further regulation in the US and abroad. [Payne] In the absence of regulation, demand, or competitive offerings, good citizenship or uncertain predictions of future demand have had varying influences. Some organizations, however, such as Microsoft, are spending

[1] Other laws having some impact include The Sarbanes-Oxley Act, Gramm-Leach-Bliley Act, and California SB 1386 (California Civil Code § 1798.82)

substantial sums in attempts to improve the security of their products. [Howard 2003] [Walsh]

Following increased awareness and unfortunate experiences, customers and users have increased demands for security, but it is unclear how much more they will be willing to pay for it or what the payoff for producers will be. Also, currently in the US, vendors are avoiding almost all liability for any damages done or expenses caused to their customers and users from software security problems. Lastly, secure software can only provide security if it is properly operated and used. Organizations can experience discouraging incidents not because of software faults but from such causes as improper settings or procedures.

Reportedly, Microsoft has found that 50% of software security problems are design flaws. [McGraw 2003] Avoiding these types of design problems requires high levels of security and design expertise. Within many software development organizations, personnel currently do not have the mathematical backgrounds or the software and security design and programming sophistication to use many of the approaches covered in the Practices section. Like the prerequisite need to have a good software process before one can have an outstanding one mentioned in the introduction, any such personnel shortcomings also must be addressed.

While for most organizations the needed substantial change will be a hard path and known practices are not perfect, the following sections in this report provide information and motivation to those willing to take the journey.

Requirements for Processes and Practices to Produce Secure Software

Overview

The previous section described the problems in producing secure software. Addressing these problems requires organizations developing and maintaining software use processes that consistently produce secure software. This section describes the key requirements for such software development processes. Follow-on sections discuss specific practices, process verification, and introducing such a process into an organization.

These process requirements do not require any particular design, development, testing, or other methods. Rather, they enumerate required characteristics – including management, and measurement support.

Process Requirements

To effectively produce secure software and provide a secure cyber infrastructure, any selected process must meet the following requirements.

- ***Coverage*** A secure process must cover the full software lifecycle from the earliest requirements through design, development, delivery, maintenance, enhancement, and retirement as well as all specialties required. Whenever necessary, the process used must be capable of economically and securely maintaining, enhancing, and reworking existing products to bring them up to an acceptable level of quality and security including, when needed, exceptionally large software and systems.
- ***Definition*** The process must be precisely defined so that it can be taught, supported, verified, maintained, enhanced, and certified and all products and all process activities must be precisely and comprehensively measured.
- ***Integrity*** The process must establish and guard the integrity of the product throughout the life-cycle, starting with the requirements and design work and including rigorous, secure configuration management and deployment. The process must also provide means to help ensure the honesty, integrity, and non-maliciousness of all the persons involved in product development, enhancement, testing, and deployment.
- ***Measures*** The process must include measures to verify that the product developers are capable of consistently and correctly using the process, that the process instructors properly train the developers to consistently and accurately use the process, that the correct process was properly used to develop the product, and that the process consistently produces secure software products. In addition to measuring all of the product work, measures are also required of all security-relevant product characteristics and properties. To perform in this way, software organizations must work to defined software engineering, security, and management goals and they must precisely measure and track their work throughout the product's lifetime. These measures must provide the data needed to estimate and plan the work, to establish team and personal goals, to assess

project status, to report project progress to management, to verify the quality of all phases of the work, to assess the quality of all products produced, and to identify troublesome product characteristics and elements. Such measures must also show how well the process was used and where and when corrective actions are needed. These measures can also be used to assess the quality of the training and coaching provided to the developers and their teams. Without such measures, it would be impossible to verify the consistent and proper use of the process or of the methods and practices it supports.

- ***Tailoring*** The process must permit tailoring and enable verification that such tailoring does not compromise the security of the resulting products.
- ***Usability*** The process must be usable by properly trained and qualified software developers, security specialists, managers, and other professionals, and it must be economical and practical to use for developing a reasonably wide and defined range of software product types.
- ***Ownership*** The process must be owned, supported, widely available, and regularly updated to reflect changing threat conditions and improvements in knowledge and technology. To have continuing confidence, all process updates must be rapidly and verifiably disseminated to all users – this in turn requires having a known set of users. Misuse must be prevented. Whenever intellectual property issues are relevant, the process must be protected from counterfeiting, copying, and other forms of piracy. While the process owner could be an individual, a corporation, or a university, it could also be a government agency, a user consortium, an open-source association, or any other established group that maintained the process consistent with these requirements.
- ***Support*** The process must be fully supported with training programs, course material, instructor's guides, supporting tools, and qualification testing. Suitable knowledge and skill training must be available for all the people who are involved either directly or indirectly. Process support must include provisions for transitioning the process into widespread use, qualifying instructors to teach the process, training specialists to verify the proper use of the process, and certifying that qualified developers produced any resulting products by properly using a qualified process.
- ***State of the Practice***. The process must include use of quality state of the practice methods for design, development, test, product measurement, and product documentation. As the state of the art and state of the practice advance, the process must enable the adoption of new methods including any required training and support.

Process Application

A process owner must enable organizations to qualify their own process specialists, to train their own software people, and to manage their own development work. Provisions must also be made to provide organizations with the skills and expertise to monitor and review their own work and to verify that their products are secure. While the training and qualification of software developers could be entrusted to educational, consulting, and using organizations, all development and product qualification work must be rigorously controlled and tracked.

In addition, the training and qualification of process instructors and process coaches and/or reviewers must remain controlled and monitored to ensure a high degree of compliance. Further, there must be provisions for training all levels of development management from the most senior executives to the development team leaders, the individual team members, and the testing and assurance staffs. This training must be consistently effective in motivating and guiding all management levels in properly introducing and monitoring the use of the process.

Provisions must exist for auditing product certifications to ensure that they are justified by the available data.

Provisions must also exist for identifying and qualifying new processes or practices that produce secure software and that are shown to meet the defined requirements for a process to produce secure software.

Process Customization

To be widely applicable, a secure development process must fit the needs of each using organization. This must permit software groups to adjust the process to use new methods, tools, and technologies and be flexible enough to support integrated development teams that include software development and all other specialties needed for the software and systems work. With multiple using organizations, an organization must – within identified limitations – be able to define the specific process details for its own use, specify the methods and technologies its people will use, and gather and supply the data needed to verify the security of the process it uses and the products it produces.

Conclusions

This section has briefly summarized requirements for a secure software development process. Such a process requires outstanding software engineering, sound security engineering, extensive training, consistently disciplined work, comprehensive data, and capable and effective management and coaching support. Software processes that meet the requirements enumerated in this section will enable qualified software groups to help secure the US cyber infrastructure.

As we will see in the next section, methods and practices are known for producing high-quality and secure software, but they are not widely practiced. Their effective application requires that software teams consistently use defined, measured, and quality-controlled processes. When they work in this way, teams can produce essentially defect-free software products that are highly likely to be secure [Davis] [Hall 2002]. Further, with such processes, software groups have the data to identify the most effective quality and security practices, to ensure that these practices are consistently followed, and to verify that the resulting products are secure.

Practices for Producing Secure Software

Introduction

As discussed in the previous sections, the problem of producing secure software is both a software engineering problem and a security engineering problem. Software engineering addresses problems such as planning, tracking, quality management, and measurement as well as engineering tasks. Security engineering addresses methods and tools needed to design, implement, and test secure systems. This section starts with a discussion of software *processes* that implement software engineering best practices, followed by a description of technical *practices* that can be used in various phases of a software development lifecycle. Thirdly, management practices are discussed. Finally, additional recommendations are made to the DHS to accelerate the adoption these processes and practices for producing secure software. Overall, the recommendations can be summarized as:

1. Start with outstanding software engineering practices.
2. Augment with sound technical practices relevant to producing secure software.
3. Support with the management practices required for producing secure software.
4. Wherever possible, quantitatively measure the effectiveness of a practice and improve.

Each section starts with a description of a practice, process, or method, and then presents any evidence of its effectiveness followed by any known usage problems.

Software Engineering Practices

Many security vulnerabilities result from defects that are unintentionally introduced in the software during design and development. According to a preliminary analysis done by the CERT® Coordination Center, over 90% of software security vulnerabilities are caused by known software defect types[2], and most software vulnerabilities arise from common causes: the top ten causes account for about 75% of all vulnerabilities.

Therefore, to significantly reduce software vulnerabilities, the overall specification, design, and implementation defects in software must be reduced from today's common practices that lead to a large number of these defects in released software. Analysis performed at the Software Engineering Institute (SEI) of thousands of programs produced by thousands of software developers show that even experienced developers inject numerous defects as they produce software [Hayes] [Davis]. One design or implementation defect is injected for every 7 to 10 lines of new and changed code produced. Even if 99% of these design and implementation defects are removed before the software is released, this leaves 1 to 1.5 design and implementation defects in every thousand lines of new and changed code produced. Indeed, software benchmark studies conducted on hundreds of software projects show that the average specification, design,

[2] The definition of a defect as used in this paper is fairly broad: a defect is anything that leads to a fix in a product. Some examples of defects include requirements defects, design defects, security defects, usability defects, as well as coding errors or "bugs". To reinforce the fact that we are not just talking about coding errors, we will use the words specification, design and implementation defects throughout this section.

and implementation defect content of released software varies from about 1 to 7 defects per thousand lines of new and changed code produced [Jones 2000].

This, along with consideration of the nature of security problems, leads to the conclusion that reducing overall design and implementation defects by one to two orders of magnitude is a prerequisite to producing secure software. To be effective, these practices should be used in a planned and managed environment.

The following processes and process models were developed to improve software engineering practices. Particular attention has been paid to those that have demonstrated substantial reduction in overall software design and implementation defects, as well as reduction in security vulnerabilities.

The Team Software Process

The Software Engineering Institute's Team Software Process[SM] (TSP) is an operational process for use by software development teams. The process has been shown to be very effective for producing near defect-free software on schedule and within budget. To date, the TSP has been used by many organizations. A recent study of 20 projects in 13 organizations showed that teams using the TSP produced software with an average of 0.06 delivered design and implementation defects per thousand lines of new and changed code produced. The average schedule error was just 6% [Davis].

The TSP's operational process definitions are based on Deming's concept of an operational definition "that gives communicable meaning to a concept by specifying how the concept is measured and applied within a particular set of circumstances" [Deming]. Operational processes provide step-by-step guidance on how to do something and then how to measure what has been done.

The SEI developed the TSP as a set of defined and measured best practices for use by individual software developers and software development teams [Humphrey]. Teams using the TSP:

1. Manage and remove specification, design, and implementation defects throughout the developed lifecycle
    a. Defect prevention so specification, design, and implementation defects are not introduced to begin with
    b. Defect removal as soon as possible after defect injection
2. Control the process through measurement and quality management
3. Monitor the process
4. Use predictive measures for remaining defects

Since schedule pressures and people issues often get in the way of implementing best practices, the TSP helps build self-directed development teams, and then puts these teams in charge of their own work. TSP teams:

1. Develop their own plans

[SM] Team Software Process, TSP, Personal Software Process, and PSP are service marks of Carnegie Mellon University.

2. Make their own commitments
3. Track and manage their own work
4. Take corrective action when needed

The TSP includes a systematic way to train software developers and managers, to introduce the methods into an organization, and to involve management at all levels.

The Team Software Process for Secure Software Development (TSP-Secure) augments the TSP with security practices throughout the software development lifecycle. Software developers receive additional training in security issues, such as common causes of security vulnerabilities, security-oriented design methods such as formal state machine design and analysis, security-conscious implementation methods such as secure code review checklists, as well as security testing methods. While the TSP-Secure variant of the TSP is relatively new, a team using TSP-Secure produced near defect-free software with no security defects found during security audits and in several months of use.

The following tables show some results of using the TSP on 20 projects in 13 organizations [Davis]. The projects were completed between 2001 and 2003. Project size varied from a few hundred to over a hundred thousand lines of new and changed code produced. The mean and median size of the projects was around thirty thousand lines of new and changed code produced. Table 1 shows schedule performance compared to results reported by the Standish Group. Table 2 shows quality performance compared to typical software projects.

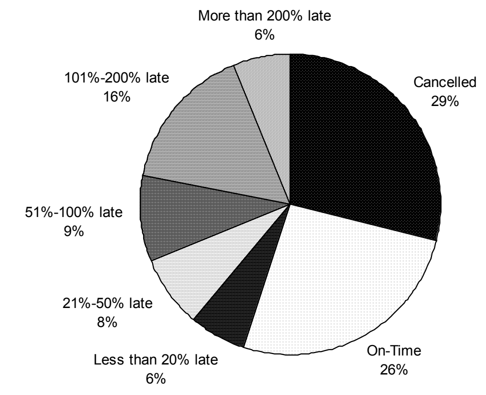
Measure	**TSP Projects**	**Typical Projects** **(Standish Group Chaos Report)**
Schedule error average	6%	
Schedule error range	-20% to +27%	

Table 1: TSP Project Results - Schedule

Measure	**TSP Projects** ***Average*** ***Range***	**Typical Projects** ***Average***
System test defects (design and implementation defects discovered during system test, per thousand lines of new and changed code produced)	0.4 0 to 0.9	2 to 15
Delivered defects (design and implementation defects discovered after delivery, per thousand lines of new and changed code produced)	0.06 0 to 0.2	1 to 7
System test effort (% of total effort of development teams)	4% 2% to 7%	40%
System test schedule (% of total duration for product development)	18% 8% to 25%	40%
Duration of system test (days/KLOC, or days to test 1000 lines of new and changed code produced)	0.5 0.2 to 0.8	NA[3]

Table 2: TSP Project Results - Quality

The difficulties with using the TSP primarily concern the initial required investment in training. To properly use the TSP, software developers must first be trained in the Personal Software Process (PSP) and must be willing to use disciplined methods for software development. The TSP cannot be introduced or sustained without senior and project management support and oversight. Finally, for most organizations, effective TSP use requires that the management and technical cultures enable rigorously performed technical work and consistent, sustained coaching, empowerment, and motivation of self-directed TSP teams.

Formal Methods

Formal methods are mathematically-based approaches to software production that use mathematical models and formal logic to support rigorous software specification, design, coding, and verification. The goals of most formal methods are to

- Reduce the defects introduced into a product, especially during the earlier development activities of specification and design.
- Place confidence in the product not on the basis of particular tests, but on a method that covers all cases

Formal methods can be applied to a few or to almost all software development activities: requirements, design, and implementation. The degree to which formal methods are applied varies from the occasional use of mathematical notations in specifications otherwise written in English, to the most rigorous use of fully formal languages with precise semantics and associated methods for formal proofs of consistency and correctness throughout development.

There is an alphabet soup of tools, notations, and languages available for use: from (alphabetically) ADL (Algebraic Design Language), a higher-order software specification

[3] This data was not available.

language based on concepts in algebra, developed at the Oregon Graduate Institute, to Z (Zed), a formal notation for writing specifications [Spivey].

Several NASA case studies describe the results of using formal methods for requirements analysis [NASA]. Benefits of using a formal specification notation such as the Z notation have been documented [Houston]. With at least one negative exception [Naur], other studies investigating the effectiveness of formal methods have been somewhat inconclusive, but tend to support a positive influence on product quality [Pfleeger].

Correctness-by-Construction

One process that incorporates formal methods into an overall process of early verification and defect removal throughout the software lifecycle is the Correctness-by-Construction method of Praxis Critical Systems Limited [Hall 2002]. The principles of Correctness-by-Construction are:

1. Do not introduce errors in the first place.
2. Remove any errors as close as possible to the point that they are introduced.

This process incorporates formal notations used to specify system and design components with review and analyses for consistency and correctness. For secure systems, they categorize system state and operations according to their impact on security and aim for an architecture that minimizes and isolates security-critical functions reducing the cost and effort of the (possibly more rigorous) verification of those units.

The Correctness-by-Construction method has produced near-defect-free software in five projects completed between 1992 and 2003, with delivered defect densities ranging from 0.75 to 0.04 defects per thousand lines of code. Two of the five projects had substantial security requirements to fulfill. The following table shows details [Hall 2004]. Table 3 presents key metrics for each of these projects. The first column identifies the project. The second identifies the year in which the project was completed. Column three shows the size of the delivered system in physical non-comment, non-blank lines of code. The fourth column shows productivity (lines of code divided by the total project effort for all project phases from project start up to completion). The final column reports the delivered defect rate in defects per thousand lines of code.

Project	**Year**	**Size (loc)**	**Productivity (loc per day)**	**Defects (per kloc)**
CDIS	1992	197,000	12.7	0.75
SHOLIS	1997	27,000	7.0	0.22
MULTOS CA	1999	100,000	28.0	0.04
A	2001	39,000	11.0	0.05
B	2003	10,000	38.0	0[4]

Table 3: Correctness-by-Construction Project Results

Almost all US software production organizations know little or nothing about formal methods; some others are reluctant to use them. First, while for many methods the actual

[4] This project has been subject to evaluation by an independent V&V organization. Zero software defects have been found, but the independent test results are not yet officially released.

mathematics involved is not advanced, these methods require a mathematically rigorous way of thinking that most software developers are unfamiliar with. Second, as with TSP, they involve substantial up-front training. Lastly, the methods require the use of notations, tools, and programming languages that are not in widespread use in industry, thus requiring substantial changes from the way most organizations produce software today.

Cleanroom

Cleanroom software engineering [Linger 2004] [Mills] [Powell] is a theory-based, team-oriented process for developing and certifying correct software systems under statistical quality control. The name "Cleanroom" conveys an analogy to the precision engineering of hardware cleanrooms. Cleanroom covers the entire life cycle, and includes project management by incremental development, function-based specification and design, functional correctness verification, and statistical testing for certification of software fitness for use. Cleanroom teams are organized into specification, development, and certification roles. Cleanroom software engineering achieves statistical quality control over software development by separating the design process from the statistical testing process in a pipeline of incremental software development, as described below.

Incremental Development. System development is organized into a series of fast increments for specification, development, and certification. Increment functionality is defined such that successive increments 1) can be tested in the system environment for quality assessment and user feedback, and 2) accumulate into the final product—successive increments plug into and extend the functionality of prior increments; when the last increment is added, the system is complete. The theoretical basis for such incremental development is referential transparency between specifications and their implementations. At each stage, an executing partial product provides evidence of progress and earned value. The incremental development motto is "quick and clean;" increments are small in relation to entire systems, and developed fast enough to permit rapid response to user feedback and changing requirements.

Function-Based Specification and Design. Cleanroom treats programs as implementations of mathematical functions or relations. Function specifications can be precisely defined for each increment in terms of black box behavior, that is, mappings from histories of use into responses, or state box behavior, that is, mappings from stimulus and current state into response and new state. At the lower level of program design, intended functions of individual control structures can be defined and inserted as comments for use in correctness verification. At each level, behavior with respect to security properties can be defined and reviewed.

Functional Correctness Verification. A correctness theorem defines the conditions to be verified for each programming control structure type. Verification is carried out in team inspections with the objective of producing software approaching zero defects prior to first-ever execution. Experience shows any errors left behind by human fallibility tend to be superficial coding problems, not deep design defects.

Statistical Testing. With no or few defects present at the completion of coding, the role of testing shifts from debugging to certification of software fitness for use through usage-based statistical testing. Models of usage steps and their probabilities are sampled to generate test cases that simulate user operations. The models treat legitimate and intrusion usage on a par, thereby capturing both benign and threat environments. Usage-based testing permits valid statistical estimation of quality with respect to all the executions not tested and tends to find any remaining high-failure-rate defects early, thereby quickly improving the MTTF of the software. Because fewer defects enter test, Cleanroom testing is more efficient. Historically, statistical testing has been a tool to predict reliability, not security.

Cleanroom Quality Results

The Cleanroom process has been applied with excellent results. For example, the Cleanroom-developed IBM COBOL Structuring Facility automatically transforms unstructured legacy COBOL programs into structured form for improved maintenance, and played a key role in Y2K program analysis. This 85-KLOC program experienced just seven minor errors, all simple fixes, in the first three years of intensive field use, for a fielded defect rate of 0.08 errors/KLOC [Linger 1994].

Selective application of Cleanroom techniques also yields good results. For example, as reported in [Broadfoot], Cleanroom specification techniques were applied to development of a distributed, real-time system. Cleanroom specifications for system components were transformed into expressions in the process algebra CSP. This allowed use of a theorem prover or model checker to demonstrate that the resulting system was deadlock-free and independent of timing issues. The resulting system consisted of 20 KLOC of C++ which in twelve months of field use of the system, only eight minor defects were discovered; all localized coding errors easy to diagnose and fix.

A number of Cleanroom projects involve classified activities that cannot be reported upon. Overall experience shows, however, that fielded defect rates range from under 0.1 errors/ KLOC with full Cleanroom application to 0.4 defects/KLOC with partial Cleanroom application. Many code increments never experience the first error in testing, measured from first-ever execution, or in field use. Defects found have tended to be coding errors rather than specification or design problems.

Adopting Cleanroom Software Engineering requires training and discipline. Cleanroom utilizes theory-based correctness verification in team reviews rather than less-effective unit debugging – for some programmers, this switch can be an initial stumbling block. Some Cleanroom methods have been incorporated in TSP projects. Its methods of proof are performed more informally than those in Correctness by Construction and are more accessible to programmers.

Process Models

Process models provide goal-level definitions for and key attributes of specific processes (for example, security engineering processes), but do not include operational guidance for process definition and implementation – they state requirements and activities of an acceptable process but not how to do it. Process models are not intended to be how-to guides for improving particular engineering skills. Instead, organizations can use the goals and attributes defined in process models as high-level guides for defining and improving their management and engineering processes in the ways they feel are most appropriate for them.

Capability Maturity Model®s (CMMs) are a type of process model intended to guide organizations in improving their capability to perform a particular process. CMMs can also be used to evaluate organizations against the model criteria to identify areas needing improvement. CMM-based evaluations are not meant to replace product evaluation or system certification. Rather, organizational evaluations are meant to focus process improvement efforts on weaknesses identified in particular process areas. CMMs are currently used by over a thousand organizations to guide process improvement and evaluate capabilities.

There are currently three CMMs that address security, the Capability Maturity Model Integration® (CMMI®), the integrated Capability Maturity Model (iCMM), and the Systems Security Engineering Capability Maturity Model (SSE-CMM). A common Safety and Security Assurance Application Area is currently under review for the iCMM and CMMI, along with a new Process Area for Work Environment, and the proposed goals and practices have been piloted for use. All of these CMMs are based on the Capability Maturity Model (CMM®). Further information about the SSE-CMM is available at http://www.sse-cmm.org, about the CMMI at http://www.sei.cmu.edu, and about iCMM at www.faa.gov/aio or www.faa.gov/ipg. Further information is also available in materials that accompany this report.

The plethora of models and standards can be somewhat daunting (SSE-CMM, iCMM, CMMI-SE/SW/IPPD and CMMI-A, ISO 9001:2000, EIA/IS 731, Malcolm Baldrige National Quality Award, Total Quality Management, Six Sigma, President's Quality Award criteria, ISO/IEC TR 15504, ISO/IEC 12207, and ISO/IEC CD 15288). Evidence exists, however, that using process models for improving the software process results in overall reduction in design and implementation defects in the software produced [Herbsleb] [Goldenson] [Jones].

Technical Practices

Some security vulnerabilities are caused by oversights that lead to defect types such as declaration errors, logic errors, loop control errors, conditional expression errors, failure to validate input, interface specification errors, and configuration errors. These causes can be addressed to a large degree by using software engineering practices. However, other security vulnerabilities are caused by security-specific modeling, architecture, and design issues such as failure to identify threats, inadequate authentication, invalid authorization, incorrect use of cryptography, failure to protect data, and failure to carefully partition applications. Effective practices that directly address security are needed to handle these problems. Technical practices must be used within the overall context of a planned and managed process for producing secure software that plans the use of the practices, monitors their execution, and measures their effectiveness. Most, if not all, of the technical practices described here require considerable security expertise. Available expert help is recommended during all phases of the software lifecycle, especially during specification and design.

® Capability Maturity Model, CMM, Capability Maturity Model Integrated, and CMMI are registered trademarks of Carnegie Mellon University.

While many technical practices are in use today for producing secure software, very little empirical evidence exists of their effectiveness.

This section begins with a discussion of some well-tested principles for secure software development. Then, some of the better-known practices for producing secure software are briefly described. Other practices worth considering exist. The list of practices included in this subsection is not exhaustive, but is hopefully representative. Empirical or anecdotal evidence of effectiveness is noted where available.

Principles of Secure Software Development

While principles alone are not sufficient for secure software development, principles can help guide secure software development practices. Some of the earliest secure software development principles were proposed by Saltzer and Schroeder in 1974 [Saltzer]. These eight principles apply today as well and are repeated verbatim here:

1. Economy of mechanism: Keep the design as simple and small as possible.
2. Fail-safe defaults: Base access decisions on permission rather than exclusion.
3. Complete mediation: Every access to every object must be checked for authority.
4. Open design: The design should not be secret.
5. Separation of privilege: Where feasible, a protection mechanism that requires two keys to unlock it is more robust and flexible than one that allows access to the presenter of only a single key.
6. Least privilege: Every program and every user of the system should operate using the least set of privileges necessary to complete the job.
7. Least common mechanism: Minimize the amount of mechanism common to more than one user and depended on by all users.
8. Psychological acceptability: It is essential that the human interface be designed for ease of use, so that users routinely and automatically apply the protection mechanisms correctly.

Later work by Peter Neumann [Neumann], John Viega and Gary McGraw [Viega], and the Open Web Application Security Project (http://www.owasp.org) builds on these basic security principles, but the essence remains the same and has stood the test of time.

Threat Modeling

Threat modeling is a security analysis methodology that can be used to identify risks, and guide subsequent design, coding, and testing decisions. The methodology is mainly used in the earliest phases of a project, using specifications, architectural views, data flow diagrams, activity diagrams, etc. But it can also be used with detailed design documents and code. Threat modeling addresses those threats with the potential of causing the maximum damage to an application.

Overall, threat modeling involves identifying the key assets of an application, decomposing the application, identifying and categorizing the threats to each asset or component, rating the threats based on a risk ranking, and then developing threat mitigation strategies that are then implemented in designs, code, and test cases.

Microsoft has defined a structured method for threat modeling, consisting of the following steps [Howard 2002].

1. Identify assets
2. Create an architecture overview
3. Decompose the application
4. Identify the threats
5. Categorize the threats using the STRIDE model (Spoofing, Tampering, Repudiation, Information disclosure, Denial of service, and Elevation of privilege)
6. Rank the threats using the DREAD categories (Damage potential, Reproducibility, Exploitability, Affected users, and Discoverability).
7. Develop threat mitigation strategies for the highest ranking threats

Other structured methods for threat modeling are available as well [Schneier].

Although some anecdotal evidence exists for the effectiveness of threat modeling in reducing security vulnerabilities, no empirical evidence is readily available.

Attack Trees

Attack trees characterize system security when faced with varying attacks. The use of Attack Trees for characterizing system security is based partially on Nancy Leveson's work with "fault trees" in software safety [Leveson]. Attack trees model the decision-making process of attackers. Attacks against a system are represented in a tree structure. The root of the tree represents the potential goal of an attacker (for example, to steal a credit card number). The nodes in the tree represent actions the attacker takes, and each path in the tree represents a unique attack to achieve the goal of the attacker.

Attack trees can be used to answer questions such as what is the easiest attack. The cheapest attack? The attack that causes the most damage? The hardest to detect attack? Attack trees are used for risk analysis, to answer questions about the system's security, to capture security knowledge in a reusable way, and to design, implement, and test countermeasures to attacks [Viega] [Schneier] [Moore].

Just as with Threat Modeling, there is anecdotal evidence of the benefits of using Attack Trees, but no empirical evidence is readily available.

Attack Patterns

Hoglund and McGraw have identified forty-nine attack patterns that can guide design, implementation, and testing [Hoglund]. These soon to be published patterns include:

1. Make the Client Invisible
2. Target Programs That Write to Privileged OS Resources
3. Use a User-Supplied Configuration File to Run Commands That Elevate Privilege
4. Make Use of Configuration File Search Paths
5. Direct Access to Executable Files

6. Embedding Scripts within Scripts
7. Leverage Executable Code in Nonexecutable Files
8. Argument Injection
9. Command Delimiters
10. Multiple Parsers and Double Escapes
11. User-Supplied Variable Passed to File System Calls
12. Postfix NULL Terminator
13. Postfix, Null Terminate, and Backslash
14. Relative Path Traversal
15. Client-Controlled Environment Variables
16. User-Supplied Global Variables (DEBUG=1, PHP Globals, and So Forth)
17. Session ID, Resource ID, and Blind Trust
18. Analog In-Band Switching Signals (aka “Blue Boxing”)
19. Attack Pattern Fragment: Manipulating Terminal Devices
20. Simple Script Injection
21. Embedding Script in Nonscript Elements
22. XSS in HTTP Headers
23. HTTP Query Strings
24. User-Controlled Filename
25. Passing Local Filenames to Functions That Expect a URL
26. Meta-characters in E-mail Header
27. File System Function Injection, Content Based
28. Client-side Injection, Buffer Overflow
29. Cause Web Server Misclassification
30. Alternate Encoding the Leading Ghost Characters
31. Using Slashes in Alternate Encoding
32. Using Escaped Slashes in Alternate Encoding
33. Unicode Encoding
34. UTF-8 Encoding
35. URL Encoding
36. Alternative IP Addresses
37. Slashes and URL Encoding Combined
38. Web Logs
39. Overflow Binary Resource File
40. Overflow Variables and Tags
41. Overflow Symbolic Links
42. MIME Conversion
43. HTTP Cookies
44. Filter Failure through Buffer Overflow
45. Buffer Overflow with Environment Variables
46. Buffer Overflow in an API Call
47. Buffer Overflow in Local Command-Line Utilities
48. Parameter Expansion
49. String Format Overflow in syslog()

These attack patterns can be used discover potential security defects.

Developer Guidelines and Checklists

Secure software development guidelines are statements or other indications of policy or procedure by which developers can determine a course of action. Guidelines must not be confused with processes or methods; they do not provide step-by-step guidance on how to do something. Rather, they are principles that are useful to remember when designing systems.

Some universal guidelines that are common across organizations such as Microsoft, SAP, and also promoted by the Open Web Application Security Project (http://www.owaspp.org) are listed here:

- Validate Input and Output
- Fail Securely (Closed)
- Keep it Simple
- Use and Reuse Trusted Components
- Defense in Depth
- Security By Obscurity Won't Work
- Least Privilege: provide only the privileges absolutely required
- Compartmentalization (Separation of Privileges)
- No homegrown encryption algorithms
- Encryption of all communication must be possible
- No transmission of passwords in plain text
- Secure default configuration
- Secure delivery
- No back doors

Checklists help developers with lists of items to be checked or remembered. Security checklists must be used with a corresponding process to be useful. For example, when security code review checklists are used during code reviews, their use must be assured, their effectiveness measured, and they must be updated based on their effectiveness.

Code checklists are usually specific to a particular programming language, programming environment, or development platform. Sample security checklists from organizations such as Microsoft and SAP are included in the on-line reference available with this paper. References to other checklists are also provided.

Lifecycle Practices

Overview

This overview subsection is based closely on [McGraw 2004] appearing in IEEE Security and Privacy magazine and is used with permission of the author. Most approaches in practice today encompass training for developers, testers, and architects, analysis and auditing of software artifacts, and security engineering. Figure 3 specifies one set of practices that software practitioners can apply to various software artifacts produced. The remainder of this section identifies a number of existing practices and lessons.

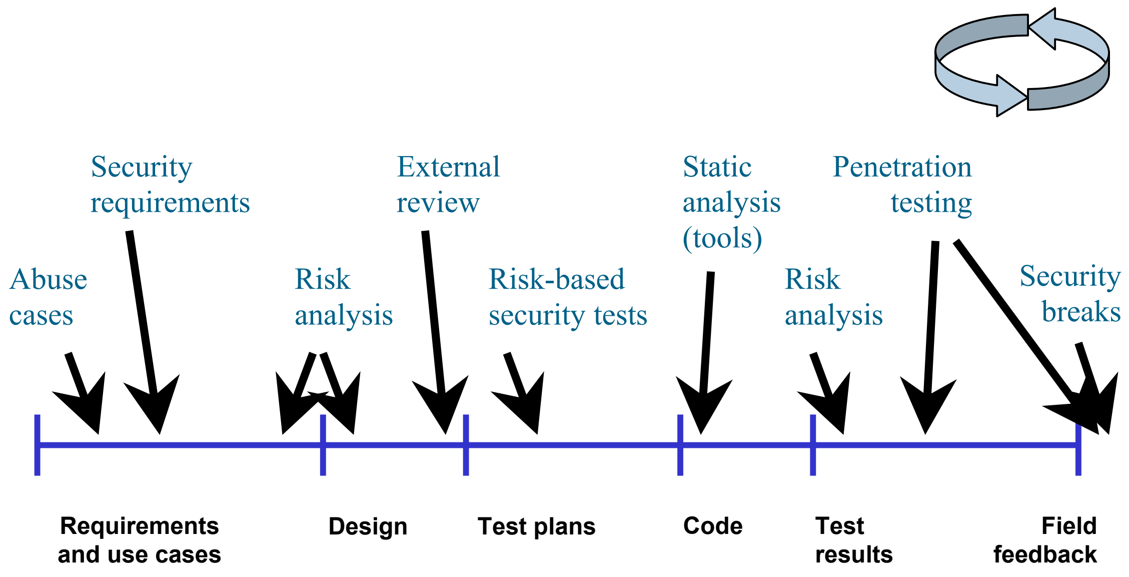


Figure 3: Software security best practices applied to various software artifacts. Although the artifacts are laid out according to a traditional waterfall model in this picture, most organizations follow an iterative approach today, which means that best practices will be cycled through more than once as the software evolves

Security requirements must explicitly cover both overt functional security (e.g. cryptography) and emergent systems characteristics and properties. One practice is *abuse cases*. Similar to use cases, abuse cases describe the system's behavior under attack; building them requires explicit coverage of what should be protected, from whom, and for how long.

At the design and architecture level, a system must be coherent and present a unified security architecture that takes into account security principles (such as the principle of least privilege). Designers, architects, and analysts must clearly document assumptions and identify possible attacks. At both the specifications-based architecture stage and at the class-hierarchy design stage, *risk analysis* is a necessity—security analysts should uncover and rank risks so that mitigation can begin. Disregarding risk analysis at this level will lead to costly problems down the road. *External analysis* (outside the design team) is often helps.

At the code level, use *static analysis tools* – tools that scan source code for common vulnerabilities. Several exist as mentioned below, and rapid improvement is expected in 2004. Code review is a necessary, but not sufficient, practice for achieving secure software because requirements, architectural, and design defects are just as large a problem. The choice of programming language also has impact and is addressed in its own subsection below.

Security testing is essential and is addressed at some length in it own subsection below.

Operations people should carefully monitor fielded systems during use for *security breaks*. Attacks will happen, regardless of the strength of design and implementation, so monitoring software behavior is an excellent defensive technique. Knowledge gained by understanding attacks and exploits should be cycled back into the development organization, and security practitioners should explicitly track both threat models and attack patterns.

Note that risks crop up during all stages of the software life cycle, so a constant *risk analysis* thread, with recurring risk tracking and monitoring activities, is highly recommended. Risk analysis is discussed at greater length below.

Programming Languages

The choice of programming language can impact the security of a software product. The best programming languages are ones where all actions are defined and reasonable, features such as strong typing are included to reduce mistakes, memory is managed appropriately, and where the use of pointers is discouraged. A language that can be formally verified, such as the SPARK subset of Ada and its associated verification tools [Barnes], would be even better. Thus languages like C and C++ have inherent characteristics that can lead to security vulnerabilities. While languages such as JAVA and C# are better for developing secure software, even better choices exist. Note that the use of a particular language does not guarantee or deny security: with care and substantial effort secure applications could in theory be written in C, and insecure applications can be written in JAVA and C#.

Tools

Several types of tools are available to support producing secure software. These range from automated tools for verification and validation of formal specifications and design, to static code analyzers and checkers. Information about automated tools for formal methods is available at http://www.comlab.ox.ac.uk/archive/formal-methods.html. Some better known code analysis tools are RATS (http://www.securesw.com/rats), Flawfinder (http://www.dwheeler.com/flawfinder), ITS4 (http://www.cigital.com/its4), and ESC/Java (http://www.niii.kun.nl/ita/sos/projects/escframe.html). The usability of static code analyzers varies. For some, their output can be voluminous (although this may reflect the poor practices used in writing the code), and the problems flagged can require human follow up analysis. For example, here is an output from a static analysis tool. This would almost certainly require a code review and maybe a design review to follow-up.

```
Input.c:5: High: fixed size local buffer
Extra care should be taken to ensure that character arrays that are allocated on the
stack are used safely. They are prime targets for buffer overflow attacks.
```

Tools used by Microsoft such as PREfast and PREfix [Bush], and SLAM (http://www.research.microsoft.com) are helping reduce overall defects. According to Microsoft, PREfix and PREfast have been very effective and caught about 17 percent of the bugs found in Microsoft's Server 2003 [Vaughan]. The Fluid project has also shown promising results (http://www.fluid.cmu.edu/). Sun's JACKPOT project (http://research.sun.com/projects/jackpot/) and is another tool under development. A number of additional tools based on compiler technology are expected to become available in 2004.

Testing

Security testing encompasses several strategies. Two strategies are testing security functionality with standard functional testing techniques, and *risk-based security testing* based on attack patterns and threat models. A good *security test plan* (with traceability back to requirements) uses both strategies. Security problems are not always apparent, even when probing a system directly. So, while normal quality assurance is still essential, it is unlikely to uncover all the pressing security issues.

Penetration testing is also useful, especially if an architectural risk analysis is specifically driving the tests. The advantage of penetration testing is that it gives a good understanding of fielded software in its real environment. However, any black-box penetration testing that does not take the software architecture into account probably will not uncover anything deeply interesting about software risk. Software that falls prey to canned black-box testing – which simplistic application security testing tools on the market today practice – is truly bad. This means that passing a cursory penetration test reveals very little about the system's real security posture, but failing an easy canned penetration test indicates a serious, troubling oversight.

To produce secure software, testing the software to validate that it meets security requirements is essential. This testing includes serious attempts to attack it and break its security as well as scanning for common vulnerabilities. As discussed earlier, test cases can be derived from threat models, attack patterns, abuse cases, and from specifications and design. Both white-box and black box testing are applicable, as is testing for both functional and non-functional requirements. An example tool that uses formal method concepts to aid testing is JTest™, a Java testing tool. The user writes pre- and post-conditions and invariants just as in formal methods using program proofing techniques. But, these are inserted by the tool as assertions and used to guide the automatic generation of tests that attempt to break them. JTest™ also attempts to generate tests to raise every possible exception.

The Fuzz testing method is another method of interest. Fuzz testing is a black-box testing method that tests software applications with random input. The method has proven effective in identifying design and implementation defects in software. More information about this testing method is available at http://www.cs.wisc.edu/. Another similar method that has proven effective for testing is the Ballista® method. The Ballista method is an automated, black-box testing method that is particularly suited to characterizing the exception handling capability of software modules. More information about this testing method is available at http://www-2.cs.cmu.edu/afs/cs.cmu.edu/project/edrc-ballista/www/. Once again, failure to pass this style of testing should reflect troubling oversights.

® Ballista is a registered trademark of Carnegie Mellon University.

System, user, and acceptance testing provides an invaluable source of data for software process improvement. Security defects found in tests are defects that escaped the entire software development process. They can be used to find, fix, and prevent future design and implementation defects of the same type, as well as similar defects elsewhere in the software. This topic is discussed further in the Qualification section of this report.

Risk Management

One might say that the most secure computer is one disconnected from the network, locked in a room, with its keyboard removed. Of course, such a machine is also useless. Since perfect security appears to be an unattainable goal with today's technologies, producing secure software becomes a question of risk management. Security risks must be identified, ranked, and mitigated, and these risks must be managed throughout the software product lifecycle.

Risk management involves threat identification, asset identification, and quantitatively or qualitatively ranking threats to assets. One ranking dimension involves determining what would happens if the risk came true. What would be the impact in terms of cost, schedule, loss of reputation, legal exposure, etc? The other ranking dimension considers the probability of the risk occurring: is the threat easily exploitable? Do many potential exploiters exist? How much effort would it take for someone to exploit the threat? Ranking risks by combining (multiplying) the impact and the probability is the common approach – indeed these are the two elements in most definitions of risk.

During requirements and design tradeoffs must be considered and addressed from a risk management perspective. Stated simply:

- User functionality – Resources required and remaining for security effort
- Usability – Security is a hassle
- Efficiency – Security can be slow and expensive
- Time to market – Internet time is hard to achieve while achieving security
- Simplicity – Everybody wants this but security can add complexity
- Quality takes care – Quality is necessary for security

Considering the nature and level of threats and applying risk management allows these to be approached rationally.

A technical method that can be used for risk analysis is threat modeling, described earlier in this report. Once the risks have been identified and ranked, design, implementation, and test decisions can be made to manage the risks.

Risks may be mitigated by the selection of various controls (specify, verify, design, and validate countermeasures to the risk), or risks may be transferred (purchase insurance against certain risks). Risk management uses a combination of these methods.

Other Considerations

There are other considerations common to most secure software development efforts. Some of these considerations are listed below:

Authentication, Authorization, Session Management, and Encryption

Authentication and authorization are common problems that face designers of secure applications. Authentication involves verifying that people are who they claim to be. The most common way to authenticate involves the use of a username and a password. Other authentication methods include biometric authentication based on voice recognition, fingerprint scans, or retinal scans. The problems faced during authentication include encryption, transmittal, and storage of passwords, session re-playing, and spoofing. Authentication should be handled using standard protocols and components where available, and requires special expertise to implement.

Authorization is about determining what resources an authenticated person has access to. There are standard authorization techniques available that should be used where possible, such as role-based access and group permissions. Applications should be deeply concerned with privilege management, especially the accumulation of excess privileges over time.

Closely related to authentication and authorization are impersonation and delegation, which allow one server entity such as a web server to assume the identity of a client entity such as the user of web browser. The main problem faced during impersonation and delegation is establishing trust in the server. Clients, therefore, should authenticate servers and limit the server's ability to act on their behalf.

Session management is of special concern. The HTTP protocol is a stateless protocol. Web servers respond to each request from a client individually, with no knowledge of previous requests. This makes interaction with users difficult. It is up to application designers to implement a state mechanism that allows multiple requests from a single user to be associated in a "session". Poor use of session variables to manage state can lead to security vulnerabilities.

Finally, encryption is of special concern for secure applications. Two areas of concern here are to use well-known encryption algorithms (no home grown ones), and the problems with key management. Entire books have been written on the subject, so all we will say here is that cryptography should be left to experts, application designs must consider what information needs to be encrypted when, and applications must properly implement sound key management practices.

An example of authentication, authorization, session management, and encryption in use is to maintain the integrity of a database. Any modification request must be: provable as coming from the identity claimed, provably unchanged in transit including not destroyed or duplicated, of legitimate content type, part of a legitimate identity-action pair (have right privileges), applied as part of an acceptable sequence of actions, responded to with a response that is provable to the requester to have come unchanged from the database management system, and the database state must remain unchanged except by controlled legitimate actions.

Accountability

With the development practices in common use today, it is difficult to distinguish between malicious code and the defect-ridden code that is normally produced. Although malicious code stands out and has a better chance of being identified when high-quality software is being produced, additional steps are needed to ensure that no software developer inserts malicious code intentionally. Without careful, rigorous control of and accountability for millions of lines of code, it is not easy to identify which lines of code have been added when and by whom. Code Signing could help make this possible. Every developer would use their private key to sign the code they produce. Therefore, every change to the software could be identified, analyzed, reviewed, and tested, instead of being put into the application without effective accountability.

Developers tend to have universal rights and authorized access to all parts of their development systems. This could lead to intentional or unintentional misuse or change of security functions or other features. Access to critical components or subsystems should be controlled. For example, nobody other than a knowledgeable person should be allowed to implement cryptographic algorithms (of course, abiding by the required laws of export and import restrictions on cryptographic software).

Code Signing and Code Access Authorizations are practices that may promote accountability, but do not address the issue of malicious code by themselves [McGraw and Morrisett].

Modifications and Patch Management

The Patch Management subgroup of the Task Force on Security across the Software Development Lifecycle is addressing issues relating to patch management. The Process Subgroup is more concerned with the process used for modifications and patch management.

As patches often get distributed as downloads from the Internet, the download process itself must be secured with all appropriate methods to guarantee integrity and authenticity.

As with development, the change process to the software system must be secure. Implementing patches, adding new features, or implementing new applications within a system requires that software be added or modified. Most organizations have concurrent instances of a development system, a test system, and a production system. Configuration and change control across all these systems becomes critical if changes are to be reflected in all instances of the system, starting from the development system, to the test system, and to production systems.

Use of Third-Party Software

The use of third-party software (COTS or open-source) poses some difficulties. Although there is no significant evidence of third-party software being less or more vulnerable, a process developing secure software must carefully consider the proper use of third-party software. Users of third-party software must have some means of identifying and categorizing the trust level of the component they are about to use. The best means of doing this would be to demand that third-party software be developed using secure development processes, and be validated using security validation methods.

Third-party software should also include a disclosure of security assumptions and limitations. A single 'security level' or 'minimum security' is very difficult to define for a software component that is being deployed in different environments with different security levels. A key piece of data to have when deciding one's level of confidence is to assess the quality and content of documentation of security limits and security assumptions. This is especially important for third-party software, but really applies to all software – customers should request it.

Management Practices

The importance of the role of management in ensuring that their organizations address security throughout the software development lifecycle cannot be overstressed. Partially for this reason, the National Cyber Security Taskforce for Corporate Governance has been formed "to consider cyber security roles and responsibilities within the corporate management structure, referencing and combining best practices and metrics that bring accountability to three key elements of a cyber-security system: people, process, and technology."

Since another taskforce is addressing this issue, the discussion here is limited to a listing of specific management practices.

- Establish organizational policies for secure software development. Policies help with codifying an organization's commitment to secure software development.
- Set measurable improvement goals for developing secure software. Since improvement without measurable goals is difficult, it is important for management to set specific, measurable goals. Examples of specific goals are to reduce vulnerabilities in delivered software by 50% as measured by number of patches released, or number of vulnerabilities reported.
- Establish leadership roles for security at the organization and at the project level. Some organizations have had success with roles such as Security Engineer, Security Analyst, and Security Architect. The Team Software Process for Secure Software Development (TSP-Secure) has a defined role of a Security Manager on each development project. The responsibilities of a security manager change during different phases of the software development lifecycle, and the security manager may not be the same person throughout the software development lifecycle, but this is the person who always focuses on security for that project.
- Provide resources and funding for needed training in software engineering practices, and security practices.
- Provide an oversight function through quality and security reviews of projects. Encourage reviews by external software security experts.

Recommendations for the DHS

Short Term

The DHS should:

- Provide incentives for using software development processes that can measurably reduce software design and implementation defects.
- Encourage and fund research to determine the effectiveness of existing best practices in measurably reducing software security vulnerabilities.

- Set quantifiable goals for reducing overall software defects and for reducing overall software security vulnerabilities.
- Encourage adoption of those practices deemed immediately useful in producing secure software.

Mid Term

The DHS should encourage and fund research:

- For secure software development processes that can measurably reduce software security vulnerabilities.
- To identify, document, and make available new security best practices that can measurably reduce software security vulnerabilities.
- To encourage evaluation of those practices that seem highly promising in producing secure software.

Long Term

The DHS should encourage and fund:

- Education and training in best processes to develop secure software.
- Education and training in best practices for development of secure software.
- Research into those practices that have early indications of being promising in producing secure software.

Qualifying Processes and Practices as Producing Secure Software

Purpose

To ensure that future software processes and practices consistently produce secure products, candidate processes and practices and the products they produce must be analyzed and tested to verify and validate that, when properly used, these processes and practices can be relied upon to produce secure products. This section describes the issues involved in such a qualification effort together with a proposed way to establish short, intermediate, and long-term efforts to provide a suitable software process and practice qualification program.

The Problems in Qualifying a Process as Producing Secure Software

For the United States cyber foundation to be secure, the software that supports that foundation must be secure. This requires that the processes and practices used to produce, enhance, and maintain that software be capable of producing secure software and that these processes and practices be properly used. When this report speaks of "secure software" it means that there is high confidence but no guarantee that the software is secure. Verifying that a software process can consistently produce secure software is challenging for at least seven reasons.

1. Security is a landscape of evolving threats. What may appear to be secure software today could be shown to be insecure tomorrow.
2. While tracking the ability of a software system to withstand attack provides some confidence that it is secure, there are no generally accepted ways to prove that it is. With current methods, we can only prove that it is not secure.
3. The number of tolerable security defects is quite low and verifying that fewer than such a small number of defects exist in a large program is extremely difficult.
4. Even if secure software had been initially developed, its deployment enhancement, repair, and remediation must not compromise its security. No generally accepted ways exist to verify that such software has retained its security properties.
5. Even after one or more processes had been shown to produce secure software, these processes and practices must remain effective when used by many people in many different software organizations and development environments.
6. An extensive and extended data collection effort would be required to obtain statistically significant evidence that a process consistently produces secure software.

7. The methods for qualifying the capabilities of the likely number of required processes and organizations would necessarily be time consuming and expensive.

The Suggested Verification and Qualification Strategy

To qualify a process as capable of producing secure software, that process must be used to develop, enhance, and/or remediate multiple software products and then those products must be tested or otherwise examined to verify that they are secure. However, while as mentioned in the Practices section a number of tools exist and they are expected to improve substantially, no available tools and techniques that can exhaustively test or otherwise analyze large-scale software products that by themselves can establish with high-confidence that no security problems exist in the specifications, design, or code.

A potential alternative to testing would be to have knowledgeable professionals inspect the software to attempt to ensure that it had no security defects. Inspections are generally used during development of new or enhanced systems and are highly effective. However, extensive inspections are not generally practical as a way to remediate the large body of existing software because of the large scale and great volume of the software currently available and the severe shortage of software professionals capable of conducting such security inspections. While this inspection approach cannot quickly or economically produce the high level of confidence desired, it is the best alternative available today.

The inspection problem is best illustrated by considering the enormous volume of material to be reviewed to verify that the design and implementation of even a moderate-sized one million line-of-code (MLOC) program is secure. Just to achieve the level of security in the best of today's widely-used software, software professionals would have to study 40 pages of source program listings for every 1 KLOC program module and miss at most one single security design or implementation defect. To achieve a ten-time security improvement, or a level of 100 such defects in a 1 MLOC program, inspectors would have to miss at most one such defect in 400 pages of source listings. However, products with 100 security defects would not likely meet any reasonable definition of security. Furthermore, for a level of 10 defects in a 1 MLOC program, the inspectors would have to miss at most one such defect in 4,000 pages of source program listings. Since this level of quality does not seem widely achievable, the inspection approach does not appear to be generally practical as a widely used verification method for legacy systems. To the extent that the security-critical code can be so designed that it is isolated into much smaller code clusters, the inspection strategy becomes more practical.

The third alternative approach is to have development teams and professionals measure and manage their software processes and products so that they improve to where the likelihood of having a single design or implementation defect in each 1 KLOC program module is less than 1 in 100. This approach is currently being used on a limited scale and suitably trained groups now routinely use these methods to produce software with an average of about 60 discovered functional design and implementation defects per MLOC or 6/100 defects per KLOC [Davis]. One third of these teams have had no defects found in their delivered products. Initial data also show that formal methods are achieving comparable quality levels [Hall 2004]. While these data are mostly for functional defects, the methods could be equally applicable to security defects, as long as the producers were able to recognize security design and implementation defects.

Security defects, however, concern security properties such as the confidentiality and integrity of a system and these generally emergent properties are not always detectable by looking at parts of the system. Current software quality methods were generally developed for functional defects that tend to be feature-oriented and somewhat localized. Thus, special analysis techniques are required and the best of these can involve disciplined development, testing, and evaluation processes and formal specification and design methods.

In light of these facts, the suggested verification strategy is as follows:

1. Have the developers and maintainers of candidate secure software products use processes and practices that have been shown to consistently produce low defect software.
2. To consistently follow such processes and practices, all producers and maintainers of such software must be adequately trained and so managed, supported, and coached, so that they can consistently maintain the required level of personal and team discipline.
3. These development teams and professionals must measure and manage their software processes so that they improve to where the likelihood of having a single design or implementation defect in each 1 KLOC program module is less than 1 in 100.
4. At the specification and design levels use the best available methods for ascertaining the emergent security properties of the software.
5. The development teams must track and analyze every security defect found in every product produced by every team member to understand why and how that defect was injected, where similar defects might remain in the product, and how to most efficiently find and fix all such defects.
6. When these candidate secure products are fielded, all newly-discovered vulnerabilities must be tracked to the product versions and the modules where they were found, the practices used to develop those modules must be identified, and the flaws in the process that permitted those defects to be entered and/or missed must be identified and a process fix developed and analyzed to ensure it has no negative impacts elsewhere. The teams and team members then must adjust their production processes both to prevent and to find all similar future defects.
7. The process includes how security defects are repaired and the fix rapidly disseminated to the product users, involving the immediate defect and all others of a similar nature that are identified as part of the process.
8. To ensure the continued integrity and increasing merit of qualified processes and practices, process data must be gathered and retained on every qualified use of the process and these data must be used by designated process reviewers to qualify these products. The resulting qualification records must include data on the

producers; who used the process, how they used the process, how they were trained and prepared to use the process, and the environment in which the process was used. Data must also be available on the steps used to produce and verify the quality and security of the product and on the training and qualification of the analysts, reviewers, and testers. These records must be maintained so that, should a qualified secure product be subsequently found to have security problems, the proper actions could be taken.

9. If it was found that the process was improperly used, the training and qualification of the producers, coaches, and designated reviewers should be assessed and adjusted. If improper process use was a persistent problem, the records of the producers, coaches, and designated reviewers should be reviewed and actions taken to revoke any qualifications that were no longer appropriate. Action should also be taken to review the process history to decide whether the process could be adequately repaired or if it should be disqualified.

Deciding One's Level of Confidence in a Product's Security

Consider all the evidence including:

1. The quality and history of the people who produced it
2. The characteristics and history of the kind of process used to produce it and its qualification level
3. The environment in which it was produced
4. Data on the quality and fidelity of use of the production process for this piece of software
5. Characteristics of the software itself and results of tests and analyses of it
6. Data on the execution history of the software itself
7. Data on the design security assumptions and security limits of the piece of software

Since this strategy must initially be implemented with unqualified processes and practices, various levels of security qualification must be used. This has been the case with the Common Criteria whose experience should be carefully reviewed for lessons including the reasons for the reputed delegable meaningfulness of its Levels 1-4 and for its reputed meaningfulness at higher Levels 5-7. The two most basic levels could be as follows.

The Initial Qualification Security Level The software process and its resulting products meet all (or most of) the requirements defined in this document and one or more of the listed best practices were used in the development work. All initial security qualifications should be for a specified and limited time and the qualification should lapse if data are not provided to demonstrate the effectiveness of the process for producing secure software.

The Fully Qualified Security Level The software process and the products it produces meet all of the requirements listed in this document and have been successfully used in a sufficient number of cases to provide a high level of confidence that the process produces secure software.

An example, in addition to the Common Criteria, is BITS – a non-profit industry consortium of 100 of the largest financial institutions in the United States that focuses on issues related to security, crisis management, e-commerce, payments, and emerging technologies. Aware of antitrust restrictions, BITS developed voluntary security criteria for software providers and a mechanism to test those products to certify compliance offering a product certification mark for those products that met the defined criteria. The BITS Product Certification Program has had limited effect in terms of changing the way the software industry develops more secure software. After having criteria available in 2000-2001, only two products are listed as being certified as of March 2004. Nevertheless, the question arises: could DHS and other government agencies support sector product certification efforts to permit critical infrastructure sectors to have security requirements without violating anti-trust restrictions?

As experience is gained and as the need arises, additional qualification levels will almost certainly be needed.

In principle, the strategy enumerated here is to improve the best existing processes, test and evaluate the products produced, qualify the processes that produced the products, and continue to improve the processes. The box on the right lists some of the factors to consider in evaluating the security of a software product. With current technology, one should not depend on any single factor, but rather consider the combined implications of all of the available evidence.

Promising Qualification Techniques

In addition to the steps above, complementary means exist for developing and using qualification evidence. Four areas currently appear sufficiently promising to warrant further study: evaluating available practices, surrogate measures, stress testing, and formal security proofs.

Evaluating Available Practices

As described elsewhere in this report, several security processes and practices are currently available that would, if widely used, significantly improve the security of software products. There are also several promising practices that appear likely to substantially improve the security of software products. To facilitate widespread use of the more effective of these processes and practices, those organizations that own and support software products that are currently under widespread security attack should test the available and most promising security practices listed in the practices section of this document. To evaluate these test results, these organizations should establish measurement programs that are consistent with that described in the *Suggested Verification and Qualification Strategy* section of this report. As data on these security tests become available, the results should be published and distributed and the DHS should urge software organizations to adopt those practices that are shown to be effective.

Surrogate Product Measures

With no way currently known to directly measure the security of a software product, identifying one or more surrogate measures may be possible. A surrogate measure of a product or process would produce data that correlated with the security properties of the products produced with the process. A potential example surrogate measure would be the number of selected types of design and code defects found during that product's system testing. Since it has been shown that a product's system test defects correlate with the number of defects found in that product by its users, it is possible that the defects responsible for security flaws would also correlate at least with selected types of system test defects [Humphrey, page 171]. With the growing volume of data on product security vulnerabilities and the potentially large volume of data available on the software production process, with proper controls for variation of circumstances such a surrogate correlation could likely be quickly ascertained.

Product Security Testing

A number of tools have been developed for static and dynamic security testing of software products. While no such tools are known to comprehensively identify all security vulnerabilities in software products, they could produce surrogate data that might correlate with at least some categories of vulnerabilities. The software industry should test any such potentially promising tools to see if they could provide surrogate data of this type.

Formal Security Proofs

Another possibility is formal analysis and proofs regarding security properties. This approach is advocated in the Common Criteria and long in use in the research community. It is rare but not unknown in software production practice [Hall 2002]. While formal techniques for dealing with security properties exist and should be used where appropriate, these methods are similar to many other software methods in that their effectiveness depends on the skill and discipline of the practitioner.

Recommendations for Department of Homeland Security on Software Process Qualification

While a completely satisfactory solution to these verification and qualification problems will likely not be available for several years, there are a number of immediate steps that could be taken to significantly improve the situation. These steps are described in the following sections on short-term, intermediate-term, and long-term recommendations.

Short-Term Recommendations

The three short-term recommendations are:

> The DHS should issue a recommendation that all organizations developing software adopt as rapidly as possible those practices currently deemed in this report to be immediately useful for producing and deploying secure software.
>
> The DHS should further request that those organizations that have software products with a significant annual volume of vulnerability discoveries conduct measured tests of those security practices deemed to be immediately useful and highly promising. This testing should follow the first eight steps listed in *The Suggested Verification and Qualification Strategy* section of this report. While all organizations should be encouraged to test the suggested methods, only those with a significant vulnerability history would likely have the data necessary for a statistically sound before-and-after verification of security practices.
>
> Organizations with suitable data should be asked to work with USCERT to determine if the number of selected types of system test defects in a product are a useful surrogate measure for the number of security vulnerabilities subsequently found in that product.

Intermediate-Term Recommendations

The three intermediate-term recommendations are:

> The DHS should launch a measurement and evaluation program to determine if any available tools or testing methods could be used to generate surrogate data that indicate the relative security of a software product.
>
> The DHS should assess the rate of improvement in the security of the US cyber infrastructure and work with the US software industry to define the measures and establish measurable security goals. It should then track performance against these goals on an annual basis.
>
> The DHS should initiate a qualification program to measure and evaluate software products and to qualify the software practices, processes, people, and organizations that produced them as capable of producing secure programs. It should also establish the criteria and practices to qualify program products as having met all of the conditions to be qualified as secure. This program should, over time, establish one or more qualification levels that are consistent with the degree of verification available and achieved by the qualified entities.

Long-Term Recommendations

The three long-term recommendations are:

> The DHS should track and assess the measurement and analysis programs recommended in this report and qualify those processes and methods found to be highly effective at producing secure software products.
>
> The DHS should encourage and fund research to identify, document, and make available further security software production processes, testing tools, and best security design and implementation practices. Several such potential processes and best practices are listed in the practices section of this report.
>
> The DHS should encourage broader coverage of security issues and practices in all computer-related academic teaching and research programs.

Organizational Change

For organizations desiring to improve their ability to produce secure software, this section discusses the many issues involved in organizational change and the body of knowledge and techniques addressing them. As previously stated, the cost in terms of resources and time, and the required organizational courage and discipline needed, can be discouraging. Undoubtedly, however, the capability to produce secure software necessitates an organization introduce, use, and continually improve software processes, technology, practices, and products. Consequently in this section, we address the issues involved in such organizational change.

There is a wealth of scholarly and popular press literature describing the challenges of organizational change, – a search of Amazon.com for "organizational change" yields over 32,000 results. These books describe techniques and experiences – the costs of failed change efforts and the considerable payoffs from successful ones. Such lessons learned are a good place to learn about organizational change, but where does one start among these 32,000? What we present here is a summary of some of the more notable and proven approaches to accelerate the adoption and improvement of the processes and practices and recommended references related to them.

Two conditions must exist before organizational change.

1. Commitment to the change

2. Ability to change

Without both, even a great technology will not be adopted. On the other hand, once the underlying issues are understood, people, teams, and organizations often participate favorably in a well crafted approach. In the following sections, we layout the issues and discuss some of the "right" approaches. The interested reader can find a wealth of detailed information in the references provided below.

What to expect

Figure 4 depicts a typical cycle of change for an organization undergoing the introduction and use of new processes or practices [Weinburg]. The phase "Old Status Quo" in Figure 4 denotes the situation prior to attempted change. Here, processes are working – for good or bad. In the scenario of major process improvement that we are discussing here, a new stage, "Instability" begins when someone introduces a novel idea for improvement, involving significant changes in day-to-day practices and behaviors. If it does not handle this stage appropriately and carefully, an organization is likely to abandon its improvement effort. The effort needs visible support by management and a team of change agents with the skill for steering the effort through its ordeals. Key activities of the change agents and management in this stage include listening, demonstrating empathy, being helpful, and providing plentiful amounts of consistent information addressing individuals' and groups' concerns. Without these, individuals – and organizations – can easily return to the "Old Status Quo". Management can easily see efficiency being adversely affected as people struggle to learn and incorporate new practices and processes. They must recognize this, however, as a natural side effect of the initial learning process – the performance payoff will be realized in later stages.

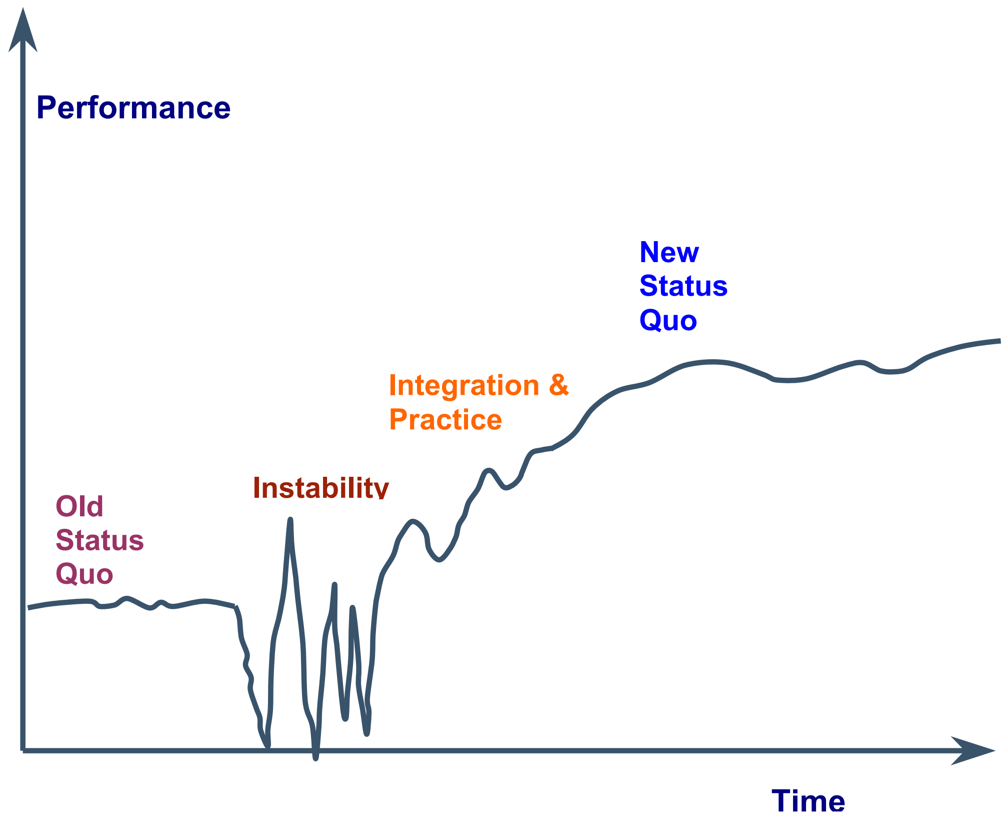


Figure 4 -Representative cycle of organizational change showing how performance can be impacted by the introduction and use of new practices

Organizations that make it through "Instability" move into a stage of performance known as "Integration and Practice". With this stage, performance begins to improve, but patience is still essential. Personnel have overcome their initial uncertainties, and start to improve their use of the new process or practice. Management needs to ensure an environment for continuing improvement with people allowed to not know everything about the new process or practice, and their questions encouraged and answered.

Finally, the organization moves into the stage of the "New Status Quo". Practice brings still more improvement, appropriate information flows through the organization, and overall, the new practices and processes are in place and working well. To optimize performance, management must grant people permission to be honest, and to explore and improve their newly acquired skills. After a period of stability and incremental improvement the organization will be ready to undertake its next major change.

These stages clearly show what all experienced change agents know – change is not instantaneous except in the most trivial of cases. Significant improvement comes from acceptance by individuals and significant change in their day-to-day behaviors, and significant change takes time, persistence, flexibility, and special skills.

Tools for Change

Successful change agents rely on a body of knowledge and a suite of techniques for supporting the movement of the organization through the stages of Figure 4. The change agent's toolkit should include, at a minimum, a thorough understanding of the following components of this body of knowledge and insight into how to use them:

- Adopter type categories and how to use them in organizational change [Rogers] [Moore 2002]
- Characteristics of adoptable technologies [Rogers]
- Stages of learning and commitment [Patterson]
- Factors of adoption [Fichman]
- Value networks [SEI]

The basic premises underlying these concepts are

- People respond differently to change
- Successfully adopted changes tend to exhibit a similar set of characteristics
- Learning and commitment to new practices follows a predictable pattern of stages
- People move through these stages at different speeds
- Change involves a network of influencers and stakeholders each of whom must individually understand and be prepared to support their role in the process.

Several additional references are included for those that wish to explore further. Introductory books are [Kotter] and [Beitler]. Intermediate books are [Christensen] [Moore 1999] and [Fench]. [Senge] is more advanced but still accessible. Coming from the study of technology transfer but quite encompassing, [Rogers] is a classic.

Organizational change is challenging, but with the right skills and approach to the change process the pay off can be quite substantial – and the same can be true along the path towards secure software.

Recommendations

In the prior sections of this report, we noted the problem of producing secure software is both a software engineering problem and a security-engineering problem. The principles for producing secure software have been known for some time. Many people involved with producing secure software are aware of the principles, as well as the practices described in this and other documents. Why do people not follow these principles and not use these practices consistently? In addition to principles and practices, a need exists for operational processes that help apply these principles in practice, provide a supportive infrastructure and environment, and a measurement system to manage and control both security pursuing processes and secure products.

As the Software Process Subgroup considered the seemingly unconnected facts on the requirements for and the capabilities of processes to produce secure software, a path to be recommended emerged. The Software Process Subgroup has confidence that following this path could lead to producing more secure software, and as a byproduct, more reliable software. The recommendations that constitute the path are highlighted in boldface. The recommendations are clustered according to the expected timing of their implementation into short-term, mid-term, and long-term recommendations.

Short-Term Recommendations

First, a very low design and implementation defect rate software production process is a necessity. As described earlier in this report, such processes exist today. Not surprisingly, they tend to have characteristics that are substantially different from the software development processes in common use. The answer is not to just keep doing more of the usual. Therefore, to start along the path, every organization desiring to produce secure software, whether a software vendor, an organization developing software for internal use, or developing open source software, should **use a process that can predictably produce software with very low specification, design, and implementation defects – less than 0.1 specification, design and implementation defects per thousand lines of new and changed code delivered.**

Given a process that produces high quality software, the next recommendation is to **make risk management central to decision making**. This requires security expertise that covers all security aspects of the system under development. Since the security expertise must be broad and deep, expert help may be needed to identify and manage security risks.

The next areas of concern are product specification and design. Security must be an integral consideration during product specification and design. **Apply formal methods to specification and design of security aspects.** Define the security properties of the software. **Analyze and review specifications and designs for security**. A key element of making this feasible is to design the software so security critical aspects are concentrated to a limited portion of the software. **The design should be as simple as possible** – possibly sacrificing efficiency – and must be restricted to structures and features that are "safe" and preferably can be analyzed. The design must not assume that the software cannot be broken and should **ensure defense in depth or tolerance**. This and other **security principles should be given close attention**.

A programming language with significantly fewer possibilities for mistakes than C or C++ should be used where possible. The programming language should be fully defined, catch all possible exceptions, and have other mistake reducing characteristics such as strong typing – and preferably safe typing.

Static analysis should be used to find known kinds of coding defects. Over time this analysis should become compulsory.

Security testing must be performed including serious attack efforts. Testing should take advantage of formal specifications and design.

Using suitable consideration, **software producers should also adopt other practices deemed useful in this report.**

Lessons can be learned from the characteristics and causes of security vulnerabilities found throughout development, testing, and after release. Organizations can improve from their own and from others' experiences, good or bad. **The products produced and process used must be constantly monitored and root causes of defects determined and reduced.**

To have a reasonable chance of success, top management, indeed management at all levels, must have a sustained and focused priority of producing secure software. Adequate resources must be available, outside expertise must be there when needed, and a quality culture must be sustained. Trained, motivated, persistent, disciplined, proficient and trustworthy personnel follow an agreed to plan and measure progress.

The recommendations so far mainly address development of new software. Just as important are practices and processes for maintenance, fixes, and patch release and management. Of particular concern are configuration management processes. **Changes to existing software should follow a rigorous change and configuration control process.**

Another area of concern is the use of COTS or open-source software. There are no significant studies that show open-source and COTS software has fewer or more security vulnerabilities. Very often, purchasers and administrators of a software product are not aware of the use of third-party software in the products they are using. Thus, they may not know that they have a security issue if the producers of the third-party software issue a warning or patch. In addition, producers of a software product may rely on the quality of third-party software without ensuring the process that produced it was adequate.

Thus **software vendors should require that third-parties developing software adopt processes and practices deemed useful in this report, or software vendors must validate third-party software before incorporating it into their products. At a minimum, they should disclose what third-party software, including open source software, is used in their product.**

Software vendors should produce a security guide or document listing the current assumptions and level of security features used such as password enforcements as well as recommendations on how this should be configured or could be possibly enhanced.

Finally, given the short amount of time that this taskforce had to write this report, only a limited number of participants could be reached to provide input – even within the organizations involved. Certainly, more knowledge and experience exists and should be utilized.

Thus the **DHS should**

- **encourage every software organization**, whether a software vendor, an organization developing software for internal use, or developing open source software, **to adopt as rapidly as possible processes that produce software that has almost no specification, design, and implementation defects**
- **encourage software organization to incorporate in-depth security expertise in their software development lifecycle**
- **request that those organizations that have software products with a significant annual volume of vulnerability discoveries conduct measured tests of those security practices deemed to be immediately useful and highly promising.**
- **ask organizations with suitable data to work with USCERT or other entity such as IT-ISAC to determine useful surrogate measure for the number of security vulnerabilities found in that product after product release.**
- **identify additional individuals and organizations working on processes to produce secure software, and request they review this report and suggest enhancements.**

Mid-Term Recommendations

For many, the short-term will not be enough time to achieve adequate levels of security. Software producers should continue to relentlessly improve the security of their products and their processes with emphasis on specification and design. Much current software will never have good security properties without substantial redesign. **Software producers must recognize systems with unacceptable architectures and designs and re-architect and redesign them with proper characteristics for security, using quality software development processes.**

This report has proposed the requirements for a process for producing secure software, as well as qualification and verification of both the process used to develop a product, as well as the product itself. However, research and experience data are needed to further validate these, as well as to get knowledge about the appropriateness of different methods.

The DHS should launch a measurement and evaluation program to determine effectiveness of secure software development processes, leading to certification of processes deemed to be capable of producing secure software.

The DHS should ask USCERT or other entity such as IT-ISAC to work with software producers and others to evaluate process and practice benchmarks to establish a baseline against which improvement could be measured.

The DHS should assess the current state of the US Cyber software infrastructure, work with the software industry to establish measurable security goals on an annual basis, and track performance against these goals.

Long Term Recommendations

Longer term recommendations all involve the DHS, and have been categorized as follows:

Certification

Certification programs like Common Criteria and ITSEC to some degree address verification of released software. Levels 5, 6, and 7 of the Common Criteria have a desirable emphasis on showing the design provides the desired security properties. It could be said that they even verify a subset of the software process as making use of methods deemed to produce lower defect software. However, the fact remains that these levels are rarely used and security incidents are increasing, not decreasing.

Thus the DHS should track and assess the measurement and analysis programs recommended in this report and **the DHS should initiate certification of those processes and methods found to be highly effective** at producing secure software products.

Education and Training

Today, most universities offering courses in security tend to focus on research oriented subjects such as cryptography, and concentrate mainly on the theory of security properties such as confidentiality and integrity. While this is good for future researchers, more emphasis is needed to train and educate at the practitioner level. Even programs that focus on the practitioner, such as those offered by some community colleges, are sometimes hit-and-miss. For example, buffer overflow prevention might or might not be taught in a programming class. A holistic approach to secure software development for practitioners is rarely found.

The Software Process Subgroup endorses the Education Subgroup recommendation that the **DHS and others should encourage and fund universities** teaching computer science or closely related subjects **to offer courses and do research in Secure Software Development Processes**. It should particularly move to enhance existing programs in this area. Analogous to medical schools where practicing medical doctors teach medical students, software security experts and practitioners should help teach computer science and software engineering students. These teachers could be recruited from organizations and companies using certified processes to develop secure software.

Accountability

The DHS should work with selected software producers to conduct experiments in implementing code-based authorizations. The purpose of the experiment would be to determine the effectiveness of limiting software developer ability to modify any part of a system, thus limiting unintended or malicious damage to critical components of the system.

The results of both experiments and experiences should be analyzed and should be made publicly available.

Evaluating New Technologies

Software technologies and applications have changed significantly in the past ten years and will do so again in the next ten. Islands of systems within a company are now connected via the Internet to systems of other companies. Firewalls, which were the preferred method to protect systems from possibly malicious access live under the paradigm that everybody within the firewall is “good” and everybody outside is “bad”. With the possible increasing use of web services, this paradigm will vanish and will need to be enhanced or replaced by web services security, perhaps involving SAML tickets and XML encryption.

This is just one example of the rapidly changing landscape in software security. New technologies will bring new opportunities as well as new challenges. New technologies will almost certainly impose ever more stringent requirements on tolerable software design and implementation defects and the processes to produce secure software. **A coordinated, sustained, ongoing effort will be needed to study the impact of new technologies on software processes for producing secure software and on legacy products.**

Given the limited time the task force had to write this paper, it is almost certain that the task force ignored other available development processes and best practices for producing secure software. Therefore, the **DHS should encourage and fund research to identify, document, and make available further security software production processes, testing tools, and security design and implementation practices, as well as other development practices for secure software**. Several such potential practices are listed in the “Practices” section of this report.

Conclusion

A path exists towards producing secure software. A few organizations are quite well along this path and show that traveling it is possible. The path involves:

1. Using an outstanding, exceedingly low-defect software engineering process and relentlessly improving it while recognizing security properties are emergent properties of systems and the central place of requirements and design
2. Incorporating sound, in-depth security expertise, practices, and technology
3. Providing the expert management to bring the resources, organization, discipline, flexibility, and persistence
4. Continuing to relentlessly improve

The Software Process Subgroup has confidence that following this path will lead to producing more secure software.

References

[ACM] *ACM Transactions on Information and System Security*, Association for Computing Machinery.

[Anderson] Anderson, Ross J., *Security Engineering: A Guide to Building Dependable Distributed Systems*. John Wiley and Sons, 2001.

[Barnes] Barnes, John. *High Integrity Software: The SPARK Approach to Safety and Security*, Addison Wesley 2003

[Beitler] Beitler, Michael A., *Strategic Organizational Change Practitioner Press International*; January 17, 2003.

[Boehm], Boehm, Barry, and Richard Turner, *Balancing Agility and Discipline: A Guide for the Perplexed*. Addison-Wesley 2003.

[Broadfoot] Broadfoot, G. and P. Broadfoot, "Academia and Industry Meet: Some Experiences of Formal Methods in Practice," *Proceedings of the Tenth Asia-Pacific Software Engineering Conference*, Chiang Mai, Thailand, December 2003, IEEE Computer Society.

[Bush] Bush, W.R., J.D. Pincus, and D.J. Sielaff, "A Static Analyzer for Finding Dynamic Programming Errors," *Software Practice and Experience*, vol. 30, June 2000

[Christensen] Christensen, Clayton M., *The Innovator's Dilemma*. HarperBusiness; January 7, 2003.

[Common Criteria Part 1] Common Criteria Project, *Common Criteria for Information Technology Security Evaluation Part 1: Introduction and general model, Version 2.1*, CCIMB-99-031, August 1999.

[Common Criteria Part 2] Common Criteria Project*, Common Criteria for Information Technology Security Evaluation Part 2: Security Functional Requirements, Version 2.1. CCIMB-99-031*, August 1999

[Davis] Davis, Noopur, and Mullaney, Julia, "The Team Software Process in Practice: A Summary of Recent Results," Technical Report CMU/SEI-2003-TR-014, September 2003.

[Deming] Deming, W. Edward. *Out of the Crisis*. Cambridge, MA: MIT Center for Advanced Engineering, 1986.

[Fench] French, Wendell L. *Organization Development and Transformation: Managing Effective Change.* McGraw-Hill/Irwin; 5th edition July 13, 1999.

[Fichman] Fichman and Kemerer, "Adoption of Software Engineering Process Innovations: The Case of Object Orientation," *Sloan Management Review*, Winter 1993, pp. 7-22.

[Goldenson] Goldenson, Dennis R. and Gibson, Diane L. "Demonstrating the Impact and Benefits of CMMI", Special Report CMU/SEI-2003-SR-009, The Software Engineering Institute, Carnegie Mellon University, 2003

[Hall 2002] Hall, Anthony, and Roderick Chapman, Correctness by Construction: Developing a Commercial Secure System, *IEEE Software*, January/February 2002, pp.18-25.

[Hall 2004] Hall, Anthony, and Rod Chapman. "Correctness-by-Construction.". Paper written for Cyber Security Summit Taskforce Subgroup on Software Process. January 2004.

[Hayes] Hayes, W. and J. W. Over, "The Personal Software Process (PSP): An Empirical Study of the Impact of PSP on Individual Engineers." CMU/SEI-97-TR-001, ADA335543. Pittsburgh, PA: The Software Engineering Institute, Carnegie Mellon University, 1997.

[Herbsleb] Herbsleb, J. et al. "Benefits of CMM-Based Software Process Improvement: Initial Results." CMU/SEI-94-TR-013, Software Engineering Institute, Carnegie Mellon University, 1994.

[Hogland] Hoglund, Greg, and Gary McGraw. *Exploiting Software: How to break code*. Addison-Wesley, 2004

[Houston] Houston, I., and S. King, "CICS Project Report: Experiences and Results from the Use of Z," *Proc. VDM 1991: Formal Development Methods*, Springer-Verlag, New York, 1991.

[Howard 2003] Howard, M., and S. Lipner, "Inside the Windows Security Push," *IEEE Security & Privacy*, vol.1, no. 1, 2003, pp. 57-61.

[Howard 2002] Howard, Michael, and David C. LeBlanc. *Writing Secure Code, 2nd edition*, Microsoft Press, 2002

[Humphrey 2000] Humphrey, Watts S. *Introduction to the Team Software Process*, Reading, MA: Addison Wesley, 2000.

[Humphrey 2002] Humphrey, Watts S. *Winning with Software: An Executive Strategy*. Reading, MA: Addison-Wesley, 2002.

[IEEE] *IEEE Security and Privacy* magazine and *IEEE Transactions on Dependable and Secure Computing.* Institute for Electrical and Electronics Engineers Computer Society.

[ISO] International Standards Organization, *International Standard ISO/IEC 15408-3:1999 Information technology – Security techniques – Evaluation criteria for IT security.*

[Jacquith] Jacquith, Andrew. "The Security of Applications: Not All Are Created Equal." At Stake Research. http://www.atstake.com/research/reports/acrobat/atstake_app_unequal.pdf

[Jones] Jones, Capers. *Software Assessments, Benchmarks, and Best Practices*, Reading, MA: Addison-Wesley, 2000.

[King] King, Steve, Jonathan Hammond, Rod Chapman, and Andy Pryor "Is Proof More Cost-Effective Than Testing?" *IEEE Transactions of Software Engineering*, VOL. 26, No. 8, August 2000.

[Kotter] Kotter, John P., *Leading Change.* Harvard Business School Press; 1st edition January 15, 1996.

[Leveson] Leveson, Nancy G. *Safeware: System Safety and Computers,* Addison-Wesley, 1995.

[Linger 1994] Linger, Richard. "Cleanroom Process Model," *IEEE Software,* IEEE Computer Society, March 1994.

[Linger 2004] Linger, Richard, and Stacy Powell, "Developing Secure Software with Cleanroom Software Engineering". Paper prepared for the Cyber Security Summit Task Force Subgroup on Software Process, February 2004.

[McGraw 2003] McGraw, Gary E., "On the Horizon: The DIMACS Workshop on Software Security", *IEEE Security and Privacy*, March/April 2003.

[McGraw and Morrisett] Gary McGraw and Greg Morrisett, "Attacking Malicious Code: A report to the Infosec Research Council", submitted to IEEE Software and presented to the Infosec Research Council. http://www.cigital.com/~gem/malcode.pdf

[McGraw 2004] McGraw, Gary, "Software Security", *IEEE Security and Privacy*, to appear March 2004

[Mills] H. Mills and R. Linger, "Cleanroom Software Engineering," *Encyclopedia of Software Engineering, 2nd ed.*, (J. Marciniak, ed.), John Wiley & Sons, New York, 2002.

[Moore 1999] Moore, Geoffrey A., *Inside the Tornado : Marketing Strategies from Silicon Valley's Cutting Edge.* HarperBusiness; Reprint edition July 1, 1999.

[Moore 2002] Moore, Geoffrey A. *Crossing the Chasm*. Harper Business, 2002.

[NASA] *Formal Methods Specification and Verification Guidebook for Software and Computer Systems: Volume 1: Planning and Technology Insertion*. Available at http://www.fing.edu.uy/inco/grupos/mf/TPPSF/Bibliografia/fmguide1.pdf

[Naur] Naur, P. "Understanding Turing's Universal Machine - Personal Style in Program Description", *The Computer Journal*, Vol 36, Number 4, 1993.

[Neumann] Neumann, Peter, *Principles Assuredly Trustworthy Composable Architectures: (Emerging Draft of the) Final Report*, December 2003

[Patterson] Patterson, Robert W. & Conner, Darryl R. "Building Commitment to Organizational Change." *Training and Development Journal*, April 1983, pp. 18-30.

[Payne] Payne, Jeffery E. "Regulation and Information Security: Can Y2K Lessons Help Us?" *IEEE Security and Privacy*. March/April 2004

[Pfleeger] Pfleeger, Shari Lawrence, and Les Hatton, "Investigating the Influence of Formal Method", *IEEE Computer*, Volume 30, No 2, Feb 1997.

[Powell] Prowell, S., C. Trammell, R. Linger, and J. Poore, *Cleanroom Software Engineering: Technology and Process,* Addison Wesley, Reading, MA, 1999.

[Rogers] Rogers, Everett. *Diffusion of Innovations*. Free Press, 1995.

[Saltzer] Saltzer, Jerry, and Mike Schroeder, “The Protection of Information in Computer Systems”, *Proceedings of the IEEE*. Vol. 63, No. 9 (September 1975), pp. 1278-1308. Available on-line at http://cap-lore.com/CapTheory/ProtInf/.

[Schneier] Schneier, Bruce. *Secrets and Lies: Digital Security in a Networked World,* John Wiley & Sons (2000)

[SEI] SEI, Technology Transition Practices, http://www.sei.cmu.edu/ttp/value-networks.html.

[Senge] Senge, Peter M., *The Fifth Discipline*. Currency; 1st edition October 1, 1994.

[Spivey] Spivey, J.M. *The Z Notation: A Reference Manual, 2nd Edition*. Prentice-Hall, 1992.

[Vaughn] Vaughn, Steven J. “Building Better Software with Better Tools”, IEEE *Computer,* September 2003, Vol 36, No 9.

[Viega] Viega, John, and Gary McGraw. *Building Secure Software: How to Avoid Security Problems the Right Way*, Reading, MA: Addison Wesley, 2001.

[Walsh] Walsh, L. "Trustworthy Yet?" *Information Security Magazine*, Feb. 2003. See http://infosecuritymag.techtarget.com/2003/feb/cover.shtml

[Weinberg] The Virginia Satir change model, adapted from G. Weinberg, *Quality Software Management, Vol. 4: Anticipating Change*, Ch 3.

[Whittaker] Whittaker, James, and Herbert Thompson. *How to Break Software Security.* Addison-Wesley, 2003.

Software Process Subgroup
Task Force on Security across the Software Development Lifecycle
National Cyber Security Summit
March 2004

Edited by Samuel T. Redwine, Jr. and Noopur Davis