

WAEPPSSD Working Group 2 Report

A report on the activities of one of two groups formed during the ACSA Workshop on the Application of Engineering Principles to System Security Design. Prepared for the Working Group by William Hugh Murray, CISSP Executive Consultant, TruSecure Corporation Senior Lecturer, Naval Postgraduate School

Acknowledgments

The chairman of the working group writes this report. If such a group is to produce a useful result, the chairman must impose a certain amount of discipline. It is inevitable that the chairman's bias will influence the results. This is particularly true when the chairman is a man whose friends, never mind his enemies, describe his style as arbitrary and capricious. The problem of bias is aggravated when the same individual records the results. Therefore, I take full responsibility for everything about this report that you, dear reader, dislike.

On the other hand, it was enlightening and delightful to work with my colleagues on the working group. Such of my bias as remains is informed and reformed by working with them. I give them full credit for all of the useful ideas in this report. I thank them for the comity with which they worked, I commend them to your recognition, and I invite them to elaborate, annotate, and correct this record as they see fit.

There is nothing new in this report. The intent of the workshop was to rediscover and encourage the application of principles that engineers have used for thousands of years. We acknowledge the contribution of the engineers to our society and our own practice.

Introduction

Context

This is the report of one of two concurrent working groups in a two-day workshop. While a few participants changed from one group to the other after the first day and one or two floated between the two, most participants remained with the group with which they began.

Focus

For purposes of the discussion, the group conceded that the results that we have produced in building, using, and maintaining secure systems are not as good as those produced by such traditional disciplines as civil or aeronautical engineering in producing safe systems. At least for purposes of discussion, the participants were willing to grant that the problems that these disciplines deal with are at least as difficult as that of secure systems and that their results are better.

[Information Technology (IT) people often argue that the security problems that they confront are more difficult than the traditional engineering problems, at least to the extent that in security we have a human adversary. However, the engineer understands that he must recognize and compensate for “easily anticipated abuse and misuse.” While the man-made threats that the security designer faces may be more complex and more difficult to anticipate, similar solutions apply.]

We tried to tread very close to the title of the workshop. We were trying to identify principles, engineering principles, and engineering principles with application to security systems. We agreed to focus on engineering principles as contrasted with security principles. We also agreed to focus on systems (e.g., networks or applications) as opposed to the traditional “multi-user shared-resource” computing system. Similarly, we distinguished between security engineering and software engineering and focused on the former.

Finally, we tried to focus on principles that enable us to build secure systems with imperfect but available materials rather than simply trying to perfect our materials. Engineers build strong systems all the time with imperfect materials and imperfect processes. We looked for the principles that enable them to do so. For example, our focus was more on how to build safe systems with commercially available operating systems than on how to perfect those systems. On the other hand, the principles identified can be applied to the building of such product systems.

Process

We used the traditional workshop process of brainstorming to identify principles and then applying the filter of our focus or mission to illuminate and order those principles. In a slight variation from traditional brainstorming, in which anyone is permitted to put anything on the table, we asked that the proposer of a principle be able to give us a “non-security” example of the principle or its application.

A Caution to the Reader

The working group had an attitude; this report reflects it. The attitude was that the engineers consistently produce better results than we do. What do they know and can we use it? What can they teach us? Can we learn it?

The report draws in stark colors things that are really shades of grey. It exaggerates to make a point. It attributes all virtue to engineers and all vice to IT and security designers. This is a literary device, not necessarily even a belief, much less existentially true; it does reflect the way people talked in the meeting. The device is used at the risk that the reader may become so defensive as to miss that point. Please, dear reader, do not spend more time questioning our assertions than learning our lessons.

Requirements

Given two statements of a problem, prefer the one that allows for a solution.

The working group spent a lot of time on the expression of requirements. One would think that, after all this time, we would be very good in our understanding and expression of security requirements. The working group found that we are not nearly as sophisticated in this space as the traditional engineers. They pointed out that while the management of security is a balancing act, security requirements are often expressed in a binary manner. Engineers tell us that if we do not know how to measure it, we do not know how to recognize it when we see it.

We take special note of the paper submitted to the workshop by Amund Hunstad and Jonas Hallberg of the Swedish Defence Research Agency entitled “Design for securability – Applying Engineering Principles to the Design of Security Architectures.” This original paper points out that “no system can be designed to be secure, but can include the necessary prerequisites to be secured during operation; the aim is *design for securability*.” That is to say, it is the securability of the system, not its security, which is the requirement. We found this idea to be elegant, enlightening, and empowering. Like many elegant ideas, once identified it seems patently obvious but it is very useful.

Atomic Requirements

Engineers caution us that requirements must be atomic. When one requirement is bound to another, it eliminates possible designs, results in a requirement that is more difficult to satisfy than either alone, and moves that requirement ahead of some that neither would trump by itself.

For example, requirements statements based upon the *Trusted Computer System Evaluation Criteria* (TCSEC) often combined the requirements for isolation and mediation into a single requirement. This eliminated solutions that placed mediation in the trusted computing base but not in the kernel. The Department of Defense (DoD) consistently rejected IBM’s VM system because the access control facility ran in a protected virtual machine rather than in the control program.

Complete Requirements

One fundamental engineering principle identified by the group is that all of the requirements must be on the table. They must all be on the same table at the same time. Securability in a system is achieved at the expense of other desirable properties.

Complete requirements describe:

- The environment in which the system must operate (including natural and artificial hazards and threats)
- The market or market conditions
- The results that the system must produce (e.g., move passengers and freight, computing application, user programming)
- Performance (e.g., passenger load, range, speed, users, standard operations per unit time)
- How it must function
- Required/forbidden controls (e.g., over-ride of automated controls)

- The granularity of the controls (maximum rate of climb, number of named users or resources; controls must be sufficiently granular that one can implement the rule of least privilege)
- Things that it must not do (e.g., stall near cruise speed, leak information between users or compartments)
- Specific threats that it must resist (e.g., lightning, easily anticipated abuse and misuse, hostile use of controls)
- Cost for overcoming resistance (e.g., cost of attack)
- Impermissible failure modes and their alternatives (e.g., halt before leaking)
- Reliability
- The availability of the system (mean-time before failure, mean-time to recovery)
- Efficiency
- Maintainability (“Function may not trump maintainability and reliability.”)
- Compatibility
- How it is to be demonstrated (e.g., testing, third-party evaluation)
- Other

Notice that this list of requirements is intended to be independent of the kind of system being specified. It works equally well for safety in airplanes and securability in information systems.

We noted that requirements that address what a system must do are more useful, easier to meet, than those that address what it may not do. This is in part because it is easier to demonstrate that the former requirements are met. It is inherently more difficult to demonstrate that requirements that address what a system may not do are met. This can be compensated for in part by negative requirements that are themselves limited.

Rank Requirements in the Order That They Must Be Satisfied

The securability of a system is a requirement that is satisfied at the expense of some other requirement. However, every engineer knows that if you ask the user/buyer to classify requirements, for example, as high, medium, or low, he will classify almost everything as high and the remainder as medium. Therefore, engineers insist that the users, buyers, and other decision makers order the list. This is in part so that the user/buyer will understand the relative cost of his requirements and forces him to resolve conflicts. All requirements may be high but not all high requirements are equal.

Take for example Microsoft Windows NT and 95/98. In the former system, securability ranked higher than backward compatibility to existing applications. In the latter systems, backward compatibility ranked ahead of securability. The result was two systems, one suitable for the enterprise, but which might require that some applications be upgraded or abandoned, and one suitable for the consumer that would be less likely to break old applications.

Consider that it is much easier to satisfy securability requirements in a system with early binding and limited flexibility. If flexibility is less important than securability, then securability will be much easier to achieve. Conversely, it is much more difficult to

satisfy and ensure securability in systems where the functionality, even if not the privilege, to alter the system itself is reserved to the users.

Unordered requirements not only make all requirements appear to have the same cost, they make them appear to have no cost. Unordered requirements result in over-constrained problems, problems without real-world solutions, and problems that cannot be solved by mere money. Ordering the list forces the user/buyer to confront his (magical) expectations. User/buyers can have anything that they want but they cannot have everything that they want. Everything comes at the cost of something else. We did not make this fundamental economic rule; we cannot change it, and must live with it. One participant noted that it takes effort to work with the users and get them to make hard choices. True. On the other hand, it is easier than meeting a magical set of expectations.

Express Requirements in Measurable Terms

The group believed that it is essential that we learn to talk about “how much” securability. The tendency of security professionals to always see security as broken or inadequate is not only incapacitating but also destructive. It was noted that in security, “Good enough always beats perfect.” Said another way, “One never wants security to be too good.” The engineers insist that if one cannot measure it, one cannot recognize its presence or absence.

Securability requirements should be expressed in measurable terms (e.g., the cost to defeat them). The necessary combination of work, access, indifference to detection, special knowledge, and time required to defeat a mechanism is a useful measure of the strength of that mechanism. It is far easier to build a system to resist an unauthorized outsider than one to resist a privileged insider. Security designers are often incapacitated by the challenge of the latter when the real requirement is for the former.

Requirements as Distinct from Policy

One way in which we have tried to describe security requirements is to say, “It must enforce the policy.” The group noted that this is okay if the policy to be enforced is really a security policy (i.e., a statement of management’s intent about what is to be protected, by whom, and to what level of cost or risk). However, we have traditionally gotten into deep difficulty when we confused access control policy with security policy. For example, one anecdotal and perhaps apocryphal failure implemented the Bell-LaPadula Model as though it were security policy. The system failed because not only did it not provide over-rides, on which the model is silent, but no controls for re-labeling objects or changing a user’s privileges. (These things were done “off-line,” while the system was down, and the enforcement ineffective.) But you “could not write down.”

System Requirements Distinct from Functional or Component Requirements

By definition, system requirements tend to be more abstract than those for functions or components. The requirements for an airliner say that it must fly most of the time but those for an engine or an engine component specify the mean time before failure.

Requirements Distinct from Mechanism

Requirements must be expressed in terms distinct from the mechanism to satisfy them. For example, one should distinguish between the requirement to resist leakage of information and of mandatory access control as a means for resisting that leakage. Similarly, one should not talk of “trusted path” as a requirement but rather as simply a means to resist eavesdropping and replay attacks.

Requirements Contrasted to Evaluation Criteria

Similarly, requirements must be distinct from evaluation criteria. The TCSEC enumerated evaluation criteria. However, the criteria were often used as a short hand for requirements. It was not unusual to hear or read, “The system requires C2 security.” As a short hand that might not have been too bad. However, this expression often short-circuited the requirements process. The real securability requirements never got expressed much less balanced against the other requirements of the system.

Complete Specification

By habit and culture, engineers use a complete specification for a system. By contrast, IT developers often work from a specification that is less than complete. A complete specification includes an expression or description:

- Of how the system is to achieve the requirements
- What functions it will perform
- What it will look like
- What materials it will be built from
- What controls and interfaces it will present
- How it will fail (failure modes)
- Indications or evidence of failure
- A model or prototype
- Users or operators manual
- Assembly processes
- Testing or demonstration procedures
- Other

We noted that for traditional procedure-oriented software that the “test data” is part of the specification, that test data includes not only the inputs but also the outputs associated with each of those inputs, and that test data should be written before, rather than after, the program. In modern display-oriented, event-driven software, the specification must describe every control (e.g., button, scroll-bar, menu) that the user will see and what result the operation of that control will produce. Note that even though the controls of an airplane are intended to be somewhat traditional and intuitive, the operator’s manual and the test procedures still describe the operation of each and the intended result of its use.

Prefer Simplicity

“In anything at all, perfection is finally attained not when there is no longer anything to add, but when there is no longer anything to take away...”

-- Antoine de Saint Exupery (Aviator, Mechanic, Poet, Exemplar), “Wind, Sand and Stars”

“Simplicate and Add Lightness”

-- Design philosophy of Ed Heinemann, Douglas Aircraft (Also attributed to Igor Sikorsky)

“Make everything as simple as possible, but not simpler.”

-- Albert Einstein

“One should not increase, beyond what is necessary, the number of entities required to explain anything.”

-- William of Occam

The group recognized that engineers, indeed even philosophers and poets, have a preference for simple designs. They eschew (gratuitous or unnecessary) complexity. One need not point out that IT designers and those who secure those systems have no such preference. Engineers try to do nothing in preference to the wrong thing. They recognize that the greater the complexity of a mechanism, the greater the potential for failure. They recognize that the greater the complexity of a mechanism, the more difficult it is to know yourself that it is performing correctly or to demonstrate to others that it is doing so. Simple designs are easier to implement, demonstrate, and maintain. Engineers promote the KISS principle. They call simple designs “elegant.” They recognize that complexity causes errors and masks malice. Engineers manage “parts count” because they know that quality goes up as parts count goes down. The number of interfaces, the number of assembly steps, the opportunity for error, and the chances of failure all go down with the number or parts.

Functional Decomposition

Engineers divide otherwise complex systems into simple and atomic components. We also do functional decomposition in IT. Unfortunately, our components may be of arbitrary size and are rarely atomic.

Separation of Privileged Controls

Engineers do not mix controls intended for the exclusive use of the managers of a system with those intended for end users. They prefer intuitive controls and safe defaults. The kind of mixing of users controls and management controls that are common in IT systems are the equivalent of putting a copy of the master-switch for the building right next to the light switch in every room. This practice invites mischief and even calamity.

Safe Defaults

Engineers strive to make products “safe out of the box” even if that means they must package separately for separate applications. Engineers prefer the minimum number of

controls consistent with the application and the environment. Not only are our IT systems often not safe out of the box but require inordinate amounts of time and special knowledge to get them to a state safe for the application and environment.

Preference for Proven Processes and Materials

Engineers prefer reuse of proven designs and materials. While each design is unique, it is still based upon proven designs. For example, while each airplane cockpit is unique, the controls in it are grouped and placed in a traditional manner. In most automobiles, the control for headlamps is to the left of the steering wheel and those for the sound system in the middle of the console. While every bridge is unique, every one shares features with every other bridge. Engineers introduce novel materials and features cautiously and only after careful testing.

While we have gotten better at reuse, we still have a marked propensity to reinvent. Part of this results from a lack of experience and knowledge, part from the fact that we are an immature field, but part results from the belief that we can do it better from scratch.

Use Materials That Are Suitable for the Application

Engineers use materials with respect for the strength and limitations of those materials. An engineer does not blame his materials, his tools, or his suppliers for the failure of his design.

One may well contrast that to IT security. We consistently use the two most popular operating systems for applications for which they are not suitable, or even intended, and then whine over the inevitable results. If we are to prefer cheap and popular over robust and diverse, we must either compensate for the inherent limitations or take responsibility for the results.

Prefer Materials with Standard Interfaces

Engineers prefer products with standard interfaces. Such interfaces enhance portability and interoperability. Not only do they reduce assembly cost but they also assure a tight fit. In security terms, they resist leaks.

Even when we choose materials because they provide these interfaces, we often break them. We exploit our knowledge of what is behind the interface to avoid the limitations of the interface. When we do that we also compromise the integrity of the “joint.”

Do Not Bend Materials To Fit

Engineers use materials in the way that they are intended to be used. Where the interface presented by one component does not match that presented by another, engineers prefer redesign to a bad fit or even to a shim.

In part because the software components that we use are malleable, we often “bend to fit” without recognizing that we have weakened the system in the process.

Defense in Depth

“One ring to rule them all, one ring to find them, one ring to bring them all, and in the darkness bind them.”

-- J. R. R. Tolkien, *Lord of the Rings*

Localize, Contain, and Compensate for Risk

Engineers understand that not all components require the same strength or the same protection. Therefore, they localize the risk and the protection. Examples include tool cribs, cashiers' cages, computer rooms, firewalls and fire doors, strong rooms, vaults, and operating system kernels. As the adversary becomes stronger or the environment more hostile, they push their perimeter defenses out and move sensitive things in and down. They layer their defenses and monitor both sides of those defenses.

Avoid Single Points of Failure

Engineers avoid single points of failure. They prefer multi-engine designs to single engine. They over-engineer those single points of failure that cannot be avoided. For example, the joint between the wing and the fuselage of an airplane may be able to resist as much as one and a half times the maximum possible load.

Share the Load

Said another way, engineers prefer designs that distribute load across multiple components. All other things equal, engineers prefer designs that spread a load across multiple relatively weak members to one that employs fewer though stronger members.

Compensate for Weak Links

Security designers are all too familiar with “A chain is only as strong as its weakest link;” engineers double the chain. Engineers know the limits of their materials and compensate for them.

Fail-Safe

Systems fail safe when they can continue to function even in the face of failing components; components fail safe when their failure does not compromise the entire system. Examples include engines in multi-engine airplanes. The airplane is designed to continue to fly even when one or more engines fail. The engine is designed to contain its own failure so as not to compromise the plane.

It is frequent, not to say usual, for IT systems to be vulnerable to single point failures. For example, the failure of a single application may bring down a multi-application system. On the other hand, the engineers demonstrate that, at some cost, it is possible and desirable to employ designs that resist such failures. Such design is particularly important for the design of security systems (e.g., firewalls, or the securability aspects of sensitive systems).

Grounds for Confidence

The group recognized that engineers test both often and early. They recognize that the earlier that an error or flaw is detected, the cheaper it will be to fix it. Engineers test under load and even to destruction.

Prefer Designs That Lend Themselves to Assurance

Engineers recognize that some designs are inherently easier to test than others. For example, systems that are simple and obvious in their intent are easier to test than complex and mysterious ones. Modularized and compartmented systems are easier to test than monolithic ones.

Bind Essential Properties Early

Engineers do not include flexibility or functionality to permit owners, managers, operators, or users to alter the essential properties of a system. For example, they may not alter the load-carrying capacity of a bridge, generator, or airplane. Even major maintenance of essential properties may be reserved to the engineers. This is an essential practice, not a mere preference. One can argue whether or not a property is essential to a particular system but not whether or not essential properties may be mutable or malleable.

Therefore, where the securability of a system is considered to be an essential property of the system, there may be no privilege, capability, function, or tools available to managers, operators, administrators, or users that could alter that property or anything essential to it. For example, where the enforcement of mandatory access controls is a requirement, this policy may be bound at design time but no later than system configuration time. Once the policy is bound, not even the system managers, and certainly not the users, may alter the policy. With sufficient privileges they may be able to alter the labels of objects or the credentials of a user, but they may not alter the policy.

This means that the securability of high assurance IT systems must be bound at build time. Change of things related to this property must require physical access to the system, special knowledge, and special tools not normally available to the owners, managers, administrators, or users.

To illustrate this point, compare the AS/400 to Windows. The AS/400 is a shared-resource multi-user system and is rarely economic in other applications and environments. Securability is an essential property of this system. Unlike most other computers, it is a finite state machine. Securability is bound in hardware and micro-code. Special tools, privileges, and knowledge, usually available only to the vendor but not to users, are required to change the hardware and micro-code. The security functionality is minimally configurable by the system manager, that is, there are six possible security settings. The default setting is the most restrictive. Access rules can be created and maintained by privileged users. However, all of the controls and tools that they use are bound by the vendor and resistant to late change.

By contrast, while Windows may support a small number of users greater than one, it is often configured and used as a single user system. While it may be used to share data, that is not necessarily so. Securability is an optional property. It is an infinite state machine. The security is all in software; Windows does not exploit any security function that might be included in the hardware. Indeed, the hardware abstraction layer, often implemented in microcode, hides any such functionality from Windows. The default settings are permissive. While there are configuration wizards to assist the system manager in configuring the security for Windows, there are a very large number of possible settings. All of the controls and tools are implemented in software and vulnerable to interference and contamination by privileged users. Users are sometimes able to increase their privileges.

Early binding of essential properties dramatically increases the reliability of a system and our confidence in it.

Prefer Single-Application Non-programmable Computers for Security Applications

Much of what we think of as the “computer security problem” is rooted in the programmability of the system and the von Neumann architecture. It is this property that makes the system vulnerable to interference from its applications, users, or users. IT people often take this property to be a given rather than a choice.

Another way of saying this is that the system should be isolated from its use. Computer systems are unique among all systems in that the division between the system and its application or use is so flexible. That is, the system and its application may both be implemented in software such that the ability to interfere with one is implicit in the ability to control the other.

Note that many of our most sensitive computers, such as those embedded in automobiles or automated teller machines, are not (late) programmable.

Test Early, Often, and Independently

In manufacturing, we test after most assembly steps. Not only is it easier to localize the flaw at this time but also it is cheaper to make a repair. This is the equivalent of unit test in IT but, unlike unit test, is often, if not usually, independent of the person who built or assembled the components.

In IT, it is not unusual for us to get to final system integration before the first independent test. Flaws detected at this point are difficult to localize, expensive to correct, and are a major cause of missed schedule.

Have Rigorous Test Exit Criteria

Engineers have strict criteria that a component or system must meet before it can exit test. They know when they are through testing. Test exit criteria are separate from test criteria. These speak not to whether or not the product meets its securability requirements or

evaluation criteria, but rather to the quality and completeness of the testing process. Is testing complete? Can we rely upon the conclusions drawn from it?

In software, testing is usually terminated when the developer can no longer find any more of his own errors. This problem is aggravated by the fact that the test results are often not documented or recorded only in the developer's head. Reconciliation to this mental record is highly error prone.

Engineers usually require check-offs by supervisors that a test has been satisfactorily completed. In software, test exit should involve the agreement of at least two people that testing is complete. In security systems, test exit criteria should be explicit and granular.

Prefer Independent Evaluation or Demonstration

In IT security, we have identified a process for independent evaluation of operating systems and security-specific products. However, this process is optional, and expensive, not to say inefficient. It is time consuming and often irrelevant to the decisions that the user/buyer has to make. It involves comparison of the "security-relevant" properties, functions, and features of the system to "security evaluation criteria." The rationale behind this program is that users will require this evaluation and use it in their buying decisions, that it will add value to the product and sufficient additional sales to cover its own cost. This process works very well for simple, cheap, high-volume, and security sensitive products like smart cards or tokens. It works less well for the kind of multi-user shared-resource operating systems for which it was originally developed. Two members of the working group had been disillusioned by their experience with this process.

That said, it is clear that developer testing may not uncover all the problems and does not produce the necessary consumer confidence. Some kind of third-party testing may still be indicated. A competing model for third-party evaluation is being used by the anti-virus and firewall industries. This vendor-sponsored model compares all of the competing products to the same technology-specific criteria. This process does not produce an evaluation report but is simply pass/fail. The criteria are the same for all products but become increasingly rigorous over time. It provides the vendors with a level playing field and the buyers with independent, objective, and relevant evidence that the product meets the same minimum securability requirements as its competitors.

Design Top-Down, Implement Bottom-Up

Engineers design from the top down. That is, before they begin a highway or a bridge, they know where the end points are and what the final system will look like. On the other hand, they build the road one mile at a time and the bridge from the foundations up. Said another way, design early, build late. Experience tells us that the longer we can resist implementation, the better the result.

In part because so many of us in IT began our careers as programmers, we have a tendency to begin writing code before we know what the final system will look like. This often has a detrimental effect on the integrity of the final product. Programmers tend to

work not so much to schedule as to end dates. They believe that the earlier they start, the easier it will be to meet the end dates.

Prefer Standard Language and Terminology

Engineers benefit greatly from their tradition of common expression and documentation. Most engineers can read and execute most designs of most other engineers.

Engineering Accountability

Given two solutions to a problem, prefer the one in your own hands.

Engineers Sign Their Work

The group observed that engineers sign their work and concluded that there was probably a relationship between this signing and the quality of the results that they achieve.

Do Not Blame Their Tools, Materials, or Vendors

Implicit in this signing is that the engineers take responsibility. They do not blame their tools, their materials, or their vendors. This may be, in part, because they do not have a convenient vendor like IBM, Microsoft, or Oracle to blame. On the other hand, it is part of the professional culture.

Do Not Blame the Customer

Engineers do not blame the customer or user. Even though the airlines are going to operate and maintain the airplane, Boeing takes responsibility for safety. They continue to negotiate until they are prepared to take responsibility for the outcome.

The workshop participants swung from extreme to extreme on this issue. On the one hand, they believed that there was no system that they could design or build that the customers could not misconfigure or misuse. The reality is that there is more than enough blame to go around and trying to assign it, particularly to the other guy, does far more harm to our efforts than it does good.

Train and Supervise Novices

Engineers understand that they must participate in and guide the education and training of new engineers. They teach and they guide the curriculum of their institutions of higher learning. Their schools, for example, MIT, Georgia Tech, Purdue, and Cal Tech, are among the most respected undergraduate programs in all of academia. They use these schools to pass on the traditions of rigor and discipline that have made their reputation.

Engineers are taught the history of their profession beginning in elementary school. They are taught about the great engineering works. They are taught both the successes and failures. They study the Brooklyn Bridge and the Tacoma Narrows Bridge. They study the Space Shuttle and the Challenger. They understand that they are part of that history: most would prefer to be identified with the Brooklyn Bridge than the Tacoma Narrows.

They are taught the science of engineering, including calculation and strength of materials. They are taught the branches of engineering from military and civil to electrical, aeronautical, and space.

Every experienced engineer expects to mentor and supervise his juniors. Every junior engineer expects the direction, support, and guidance of more experienced engineers. We need not be embarrassed that our programs of higher education do not rank with the great engineering schools. We should be embarrassed that students in our field begin *de novo*, that they do not study our history or recognize our seminal papers, appreciate our successes or recognize our failures. We should be embarrassed that they do not recognize our great teachers or inventors.

Recognize Journeyman and Masters

Engineers recognize the need for experience and they recognize it when they see it. They insist upon credentials and reserve those credentials to those qualified by training and experience to exercise professional authority, discretion, and judgment. One may well contrast this to security where self-taught and untutored nineteen-year-olds claim to be peers to those with more years of training and experience than they have years of life, and where being an unsuccessful criminal is treated as a qualification to be a security system designer.

Rigorous Process

A great deal of what has been recorded above can be summarized as rigorous process. Consider these dictionary definitions of engineering: “to contrive or plan out usually with more or less subtle skill and craft” and “skillful maneuvering or direction.” Note that these are not definitions of the practice of engineering but of the word *engineering*. Because the practice of engineering is rigorous and disciplined, the word has taken on that meaning and is applied to other things. The word is synonymous with the quality of the practice. Rigor and discipline are part of their tradition and culture. The engineer does it once. He does it “right the first time.” “Measure twice, cut once,” he says.

Though the engineers produce systems every bit as creative and novel as those that we produce, they do so by means of repeated and repeatable processes that produce predictable results. While these processes have changed greatly in the last hundred years, they would not surprise an engineer from ancient times nearly so much as would the artifacts that they are used to produce.

On the other hand, IT practice has a very different reputation. In part because we are growing so fast and are in such high demand, we use inexperienced, not to say unskilled, people with a minimum of training and supervision. Because we are driven by schedule, “quick and dirty” is our mantra and “good enough” is our standard. We say of our products “not ready for prime time.”

We think that we miss a schedule because we do not produce enough units of product per unit of time. We work ever and ever faster and fall further and further behind. The real reason that we miss a schedule is that when we put it all together, it does not work or it

does the wrong thing. Note that we measure a schedule to the exclusion of measuring quality. If we measured quality, even to the exclusion of measuring a schedule, the schedule would take care of itself.

We enjoy a well-deserved reputation for encouraging complexity and mystery and for tolerating shoddy. We are better today than we were a generation ago but we have a long way to go to earn the reputation enjoyed by the engineers.

In another hundred years, most of what is done by both engineers and IT professionals will have been automated and will be done with all of the rigor and discipline that that implies. In the meantime, we need not, indeed do not have the time to, invent our own process. We need but simply emulate the example set for us by the ancients.

William Hugh Murray, CISSP
Executive Consultant, TruSecure Corporation
Senior Lecturer, Naval Postgraduate School