SAND96-1871C
CONF-960869--8

Are Safety, Security, and Dependability Achievable in Software?

Sharon K. Fletcher, Ph.D., Sandia National Laboratories, Albuquerque, NM

This work was performed at Sandia National Laboratories, which is operated by Lockheed-Martin Corporation for the U.S. Department of Energy under contract DE-AC04-94-AL85000.

Abstract

Critical software must be safe, secure, and dependable. Traditionally, these have been pursued as separate disciplines. This presentation looks at the traditional approaches and highlights commonalities and differences among them. Each can learn from the history of the others. More importantly, it is imperative to seek a systems approach which blends all three.

Introduction

The first section summarizes typical approaches to safety, security, and dependability of software. These are very cursory overviews, from the author's perspective, intended to give a flavor of how each discipline approaches the problem in general. Both traditional and emerging viewpoints are discussed. Next, lessons are extracted. These include what seems to be working well for each discipline, as well as what's not working so well. The final section suggests a systems approach incorporating all three disciplines. The appendix takes a look at real project experiences and offers further opinions of the author, in a lighthearted vein.

Traditional Approaches and Modern Deficiencies

Software Safety

A general, high level approach to safety is to identify potential hazards in a system, and to select a protection level for each, based on a combination of probability and severity of the hazard. Protection levels can range from eliminating the hazard, to reducing it, to limiting the resultant damage.

One approach to identifying hazards is to take a process view of the system. That is, to map out the normal process flow and then ask what can go wrong - what if too much or too little material is received here? what if a valve fails there? and so on. Another approach is to construct fault trees, event trees, or cause-and-effect diagrams. In any case, each hazard is analyzed for severity and probability of occurrence. Severity and probability can form two axes of a protection level

matrix, or can be multiplied to produce a single risk rating. The matrix or rating indicates the protection level that the designer should strive for.

Taking the high level analysis approach down into the software, i.e., crossing the system-software boundary with a common analysis approach, is rarely attempted. This may be because a viewpoint that made sense at a higher level is not useful within the software. However, critical portions of software code are sometimes examined with a fault tree approach, which looks for paths that can produce unsafe output.

Often, there is a high level system safety policy, or safety theme, which provides safety goals, values, and general approaches. Safety themes include such guidelines as independence to prevent common causes of failure, isolation to prevent accidental triggering of actions, and first principles which rely on the laws of nature (e.g., gravity) as failsafes.

An obvious next step should be to map the system level safety policy and design into safety requirements on the software portions of the system. In practice this is done poorly, if at all. Inadequate specification of requirements, even in the absence of safety concerns, is a major problem in the software community today. Specifying requirements related to software's role in the larger system safety design is simply not something that is well understood.

Software engineering and system safety engineering are still relatively young fields. The combination of the two, that is, software safety, is in its infancy. However, there are some general software design approaches which can make a positive, if not measurable, contribution to system safety. Examples are user interface heuristics ("the easiest response should be the safest response"), and safety kernels. The idea of a safety kernel is to map all device controls through a single software module which enforces a safety policy for the device. Leveson's book [1] provides an excellent survey of the state of the art.

Security

The security need that has most influenced the computer marketplace is the need to protect classified information. The solution to this need was defined early-on, in terms of a reference monitor, or kernel, which mediates all file accesses. This solution re-casts the problem as controlling accesses by subjects (processes, ultimately representing users) to objects (files). Users have clearance levels, and files and processes have classification levels, and the security policy is represented by two straightforward rules: no "read up", and no "write down". Clearance and classification form two axes of a risk matrix which shows which operating systems provide sufficient capability for enforcing the rules. Operating system implementations of the reference monitor are required to be correct, non-bypassable, and tamper proof. The reference monitor is described as a state

machine to make it easier to show correctness (if the system is in a secure state, show that any transition leads to another secure state).

The computer security community is currently wrestling with the insufficiency of the above solution for today's environments. First of all, the solution cannot easily be extended to distributed, networked environments in a way that maintains the tamper-proof and non-bypassible requirements. Secondly, it only addresses a small part of the modern security picture. Today, intrusions, viruses, system integrity, and denial of service are major concerns. And it is arguable whether the reference monitor ever even solved the original problem, anyway, because processes accessing files are simply too narrow a part of the problem. Finer granularity of information, covert channels, inference, traffic analysis, and other forms of information flow were all left to be dealt with outside the basic reference monitor mechanism. It seems to be a case of failing to model the entire problem, and instead addressing only that portion that could be neatly and formally modeled.

Some very basic philosophical questions are starting to be asked. See [2] and [3], for example. The historical security approach has instilled an attitude that security mechanisms can and should be pre-defined, formally modeled, and positively stated. But this may not be possible in today's environment. Although prevention has been the predominant mindset, layers of security have always been encouraged, including the ability to detect and respond when some barriers fail. Perhaps security will swing to the other extreme: totally adaptive and on-the-fly, because maybe the best that can be done is to recognize and swiftly act on intrusions, viruses, and leaks. Such ideas are a radical departure from the past.

Software development and delivery processes are another area of growing emphasis. (Note that this use of the word "process" is different than in the previous paragraphs.) The goal is to eliminate opportunities for any person to subvert the software by inserting trapdoors, substituting other code, etc.

Dependability

Correctness is a prerequisite for dependable software. Parnas [4] suggests three complementary approaches for producing correct software: process, product, and testing. Process things include personnel certification and assessments of the software development process. Product things include examining the actual software product and related artifacts via inspections, reviews, requirements tracing, formal methods, etc. Testing complements product review by exercising the software in its actual environment; this is still important because inspections and proofs necessarily make simplifying assumptions about the environment. Abrams [5] provides a good discussion of practical approaches to correctness.

The foundational tripod above could be fortified with three additional considerations. These are: manage complexity, manage change, and manage rationale. Complexity has long been understood to have an inverse relationship to correctness, yet is fast outpacing correctness techniques. It is also generally recognized that up to 90% of project effort goes into maintenance (i.e., corrections and enhancements, i.e., changes), and that heaping changes upon changes creates fragile software. And, as anyone who has modified a legacy system will attest, design rationale is usually not captured. Understanding the rationale behind design decisions is important, especially when the design reflects safety, security, and dependablility requirements; not understanding how the design meets these requirements leads to a dangerous maintenance situation.

Reliability growth strives to reduce MTTF to a consumer-acceptable level, and concentrates on testing with expected operational profiles. [6] But critical applications need a zero defects approach, as opposed to a reliability growth approach. Critical applications should use testing to uncover environmental limits, failure modes, and behaviors in unintended environments. Process and product methods should be maximized to eliminate faults early, to manage complexity and rationale, and to provide robustness to change.

There are many good ideas scattered in the literature to guide design of dependable software - such as domain modeling, abstraction/information hiding, defensive programming, fault tolerance, and database integrity principles. Some of these have to do with increasing formality and abstraction in an effort to build the right thing and build it right. And many have to do with detecting and recovering from things gone wrong.

Blending all three

Lessons from each

Each of the disciplines offers lessons for the future, both in things that have worked well, and in shortcomings. A list of lessons is offered below.

- model the entire problem (security)
- use kernels to enforce policies (safety)
- the kernel approach is insufficient (security)
- layer & use complementary approaches - e.g., fault tolerance, fail-safes, security barriers (all)
- assume hostile adversaries (security)
- assume faults (safety)
- use guiding principles - e.g., safety themes, security policies (safety, security)
- use a range of appropriate strategies, from prevention to detection (all)
- recognize software uniquenesses (safety)

A melded viewpoint

Each of the disciplines is trying to ensure that we build the right thing, build it right, and protect it appropriately, from the viewpoint of that discipline. "Protect" takes on the flavor of the discipline -- security protects from adversaries, dependability protects from faults, and safety protects from hazards. Most critical software needs to be looked at from the perspective of all three disciplines. However, what helps from one perspective may actually be detrimental from another. For example, securing against adversaries may impede availability, which is part of dependability. And even within a discipline, an approach may have both positive and negative aspects. For example, redundancy in data servers reduces the risk from a single point of failure, but also increases the risk of data inconsistencies arising.

Each discipline takes a unique perspective on the system. Starting from any one makes it difficult to do justice to all. It would be better to find a new perspective that can balance all three. That new perspective could be "correct operation", as long this is defined to mean correct with respect to not only functionality, but also with respect to the safety, security, and dependability requirements of the system. This, of course, forces more explicit statement of such requirements, which is good. A companion paper at this conference explores this suggested approach in more detail. (See "Risk Management - What About Software?" and [3].)

A Systems approach

Incorporating the three disciplines into software design, development, and maintenance demands a systems approach. The steps to be followed are:

- define requirements
- design the system
- evaluate the design relative to the requirements
- correctly implement the design
- manage the software product over its lifecycle

While this sequence of steps looks straightforward, it actually presents many challenges! First of all, it assumes that functional, safety, security, and dependability requirements can be stated. As noted earlier, software requirements specification is still an active research topic, especially as concerns the areas of safety, security, and dependability. Secondly, it assumes that tools, or at least a methodology, exist to make comparisons between an actual design and goals (requirements) for the design. So the requirements must be stated in a way that allows for some sort of metrics to be taken. (The paper "Risk Management - What About Software?" suggests that the scale for the metrics could be risk, where risk is the possibility (perhaps probability) of not meeting or maintaining certain safety/security/dependability objectives.) Lastly, correctly

implementing the design and maintaining that correctness over the lifecycle remain significant challenges in the software community.

Conclusions

The key to achieving software safety, security, and dependability lies in taking a systems engineering approach, using a viewpoint that allows equal emphasis to all three. With this as a goal, the software community should work on characterizing safety, security, and dependability requirements, and on metrics for assessing how well designs measure up to the requirements. Current best practices, such as guiding principles and layered approaches, and lessons learned, such as avoiding too narrow a problem description, can be gathered from each discipline to contribute to formation of the common viewpoint.

Biography

Dr. Sharon K. Fletcher
Software Surety Department, MS0449
Sandia National Laboratories
Albuquerque, NM, USA 87185-0449

(505) 844-2251
skfletc@sandia.gov
fax (505) 844-9641

Dr. Sharon Fletcher is manager of the Software Surety Department at Sandia National Laboratories in Albuquerque, New Mexico. She has worked in real-time software development, signal processing, decision support systems, computer networking, and computer security. Her current research interests include high confidence software, software reliability science, and software risk management.

References

1. Leveson, N. G., Safeware: System Safety and Computers, Addison Wesley, 1995.

2. Meadows, C., "Applying the Dependability Paradigm to Computer Security", Proceedings of the New Security Paradigms Workshop, August 1995.

3. Fletcher, S. K., et. al., "Software System Risk Management and Assurance", Proceedings of the New Security Paradigms Workshop, August 1995.

4. Parnas, D. L., et. al., "Assessment of Safety-Critical Software in Nuclear Power Plants", Nuclear Safety 32-2, April-June 1991.

5. Abrams, M. D. and M. V. Zelkowitz, "Striving for Correctness", Computers and Security, 14, 1995.

6. Everett, W. W., et. al., Reliability by Design, AT&T order code 010-810-105, 1990.

Appendix
Ask Surely™

(Surely Right™ is the imaginary author of a bestselling new book, "True Stories and Practical Advice", which includes a shocking expose of how software really gets built. In this column, Surely answers questions from her readers.)

Dear Surely,

I'm curious about the process leg of the process-product-testing tripod. Can you give us some examples of process issues encountered in actual software projects?

- Nervous Project Manager

Dear Nervous,
Yes I can. Here are just a few that I am aware of:

The case of the productivity improvement tools. A project planned to use several new software development tools which held promise for improved productivity, and besides, which would be fun to use. But they failed to account for the multiple learning curves on the new tools. Upon reflection they realized that too many new things at once would have a negative impact on productivity, which the project, already under a challenging schedule, could not afford.

The case of the elusive real-time bug. Phase I of a project, involving real-time data collection and decision support, was up against an elusive bug. It was most likely a subtle timing problem, but the team lacked effective real-time debugging tools to find it. The project was behind schedule and should have been starting Phase II, which added redundancy (a considerably more complex timing situation). The development team wanted to press ahead, confident that they could find the timing error in the more complex Phase II system which they could not find in the Phase I system. A review of the situation revealed the insanity.

The case of ignoring past experience. A multi-year project was producing a new generation of a product that had been built in the past. This project involved many new, unproven concepts, so that it was more like a new product than an incremental improvement on the earlier product. Comparison of their schedule with data on the previous project showed that they were being very ambitious - expecting much more development to occur in a much shorter time span than had been accomplished previously. No one really felt this was possible.

Dear Surely,
Can you give an example of security testing troubles?

- Security Officer

Dear Security Officer,
Ah, here's one I remember well.

The Case of the Meaningless Testing. Each new release of a commercial operating system underwent weeks of testing at a customer site before being installed on their secure network. This should have made security really good, right? Wrong. Test after test was performed by the customer, repetition of testing done by the vendor, checking that the basic security mechanisms worked as advertised. What should have been tested was how things would work or break IN THE ENVIRONMENT. There were real security vulnerabilities due to the context, which the extensive testing didn't even begin to explore.

Dear Surely,
Speaking of security, you seem rather critical of the reference monitor approach, which has served us well for many years. How come?

- Traditional

Dear Traditional,
I'm afraid it is human nature to solve the problems we know how to solve, measure what we know how to measure, and build what we've built before, whether or not it is what we need now. Despite the shortcomings mentioned previously, reference monitor security was such a resounding success (vendors actually built products to it!) that other security problems began being viewed as extentions of it. For example, one approach to data integrity posed a hierarchy of integrity classes (paralleling classifications) and restricted movement of data between classes, so the existing vendor mechanisms for confidentiality need only be applied in reverse to achieve integrity. Also, network security needs were naively viewed as simply needing to protect data in transit between systems, ignoring the realities like incompatible security implementations in the individual systems, and the need for network-wide policy and mechanism. I see the community really struggling now to ovecome the narrow viewpoints that have been propagated.

Dear Surely,
What's this business of formal methods, so called "proving correctness", all about? If it's really possible, why isn't software perfect now?

- Skeptic

Dear Skeptic,
Initially, the idea of "proving" software code to be correct had real appeal and seemed like a silver bullet; now the role of proofs is better understood. Emphasis has moved from code to design. For example, critical and complex elements of a design, such as a fault-tolerance scheme, might be examined in this formal way. What can be proven are assertions (which one must be smart enough to dream up) about models (which abstract away many important details) of the system (practically, only very small parts of the system) one is designing (and which still must be transformed from correct design to correct implementation). It is clearly not the whole answer, and not something than can be undertaken casually, but for really critical systems, it may be worth an investment.

Dear Surely,
I've been hearing about "model-based software engineering" and "composition." What do these mean? Have I been time-warped onto another planet?
- Just Trying to Get My Job Done

Dear Just,
Alas, the problem is that the software systems we are building are just too complex nowadays, and we're trying to build them with yesterday's processes. We're getting behinder and behinder. David Parnas tells us we need a tripod consisting of product, process, and testing. But I would add three more ingredients to that formula: manage complexity, manage rationale, and manage change! Anyway, to answer your questions ... "Composition" is an approach to building systems out of proven modules, kind of re-use+. The far out part of it is to measure quality properties of the modules and compose them into system measurements. "Model-based software engineering" is aimed at eliminating the cascade of product artifacts (one could read that as "documents"), which we find impossible to keep in sync anyway. It implies producing everything from one central model - imagine that the model is some sort of executable spec and there are really viable automatic code generators. Maybe we do need a time-warp to help us step back and look at our work from some different perspectives.

Dear Surely,
You're making my brain hurt. Come back to earth and tell me how to deal with viruses and breakins. And I'm not just worried about myself. I'm afraid the whole nation's infrastructure is at risk. I see a terrorist around every corner.
- Paranoid

Dear Paranoid,
You're right to be scared. You see, we've been duped into thinking that security can be a zero-risk, tamper-proof kind of thing. We got close to that when our world was a mainframe and we could install a reference monitor. But now our

world boundaries are wider and vaguer. Yes, there are virus and intrusion detectors, and you should use them, but they only look for what they know how to look for. Like a flu shot, they will be helpless against the next new strain. So, the key to success is an adaptive defensive ability to discern self from intruder. And, of course, the monitor has to monitor itself for corruption, too. This is sounding like a heuristic & uncertain science. Must be very disconcerting for the reference monitor crowd. I'm even starting to get your headache. Hey, what was that behind me?

Dear Surely,
You use the word systems a lot. What is systems science?

-A Real Scientist

Dear Real,
First of all, the process known as systems engineering goes something like this: we start with some defined goals for our system, and develop decision criteria against which we can evaluate alternatives for the system design. After selecting the design, we manage the implementation, and operation of the resultant system, according to the goals. Now, implicit in this is the ability to model important characteristics of the system, make measurements, and carry out some mathematically based analysis. The science part of it develops the models, formulas, and measurement techniques.

Dear Surely,
So is software a systems science?

-Still A Real Scientist
P.S. I'm considering a job change

Dear Still,
I'm going to guess that your science (is it maybe rocket science?) is based mostly on the subdivide-and-conquer scientific principle. In other words, that you build things from parts where measurements on the parts feed into formulas that give you measurements on the whole. Perhaps a part of your science is also based on probabilities, where you use textbook mathematics on probability distribution functions. Why do you believe in these formulas? Probably because they have stood the test of time and have shown their interpolative and predictive prowess for your field. And, they are most likely based on physical phenomena that have been studied for a long time.

If you decide to become a software engineer, prepare for a severe shock. It's not as easy as rocket science. It is not easy to identify important characteristics, make measurements, and analyze software systems with models and formulas. Software doesn't really fit the subdivide-and-conquer mold, nor does it fit the

probabilistic mold. Not if we're talking about building to quality and surety goals. To tell the truth, we don't even know how to state those goals. So, although we like to say we do software engineering, it's heuristic engineering, not based on much of a science yet.

Dear Surely,

You're still making my brain hurt. Stop it already. I thought you said the title of this was practical advice.

- Confused and Hurting

Dear Confused,

Forgive me. I took two CIRTs and I'm feeling a bit more stable now. Next question, please.

(Surely Right has decided to take a much needed vacation, and wants her readers to know she plans to return soon, refreshed and ready for more questions.)