

# Educational Imperatives for Software Assurance

by Cliff Berg

Ignorance is the word that best characterizes the state of application software development today.

Whenever there is a prominent failure of an application, such as a media-worthy security breach or a newsworthy software error that causes a company to lose a lot of money or face, the news reports usually depict the problems as “bugs” that should be weeded out by programmers who need to be more careful. System administrators are reprimanded for not having more layers of network security in place and the most up-to-date patches, managers are reprimanded for not being more diligent, and programmers are reprimanded for not finding all of their “bugs”.

Yet the true nature of the problem is completely missed. Even experts who deeply understand the real issues with building reliable software do not properly articulate it. For example, with regard to security failures, the message that is most often heard is that malware is the problem, and that the solution is to add more countermeasures and to avoid making certain kind of mistakes (aka, bugs). These approaches will never let users and organizations gain ground over the malware writers. It is like a husband and wife in a fundamentally broken marriage, seeing an ever increasing number of therapists.

Consider that:

1. ***The average programmer is woefully untrained***, in basic principles related to reliability and security.
2. ***The tools available to programmers are woefully inadequate*** to expect that the average programmer can produce reliable and secure applications.
3. ***Organizations that procure applications are woefully unaware of this state of affairs*** and take far too much for granted with regard to security and reliability.

I often refer to this situation as the "MUTE problem", for Methodology, Understanding, Tools, and Education. In short, we are in a state of ignorance.

As an application developer at heart it makes me almost ill to see all the attention focused in the wrong places. The core of the problem is the inadequacy of training among software developers and the complete lack of understanding in most business software development environments of what it takes to build reliable and secure software. We need to raise awareness of the issues among mainstream software developers and get information about secure and reliable software design out of their intellectual silos and into the hands of the programmers on the ground who build systems.

# Discussion

Let's consider the problems that I have listed, one by one, starting with the educational system's deficiencies.

## The Educational Environment

One could look at the curriculums of the top schools, but that would not suit our purpose, because the majority of business application programmers do not come from the top ten schools. What we are really interested in is a representative set of undergraduate academic programs in computer science. I have not done a scientific survey: I leave that for a researcher; but just to get an idea, in order to validate my suspicions, I selected two large and prominent schools that are geographically local to me: George Mason University (GMU) and the University of Maryland. These happen to be very excellent schools. For grins, I also selected my own alma mater: Cornell University. Thus, while this small sampling is not scientific, it might at least validate the possible existence of a problem, perhaps indicating that further research is warranted.

What I found was that none of the undergraduate computer science programs of these three schools require a single course in computer security, reliability, or software development methodology. George Mason has a separate degree program called Information Security and Assurance, which does require such courses, but more of GMU's computer science graduates enroll in the regular Information Systems program.

The University of Maryland offers only one undergraduate course in security, and one in cryptography, and these are not required courses. GMU offers two undergraduate courses related to security, and one undergraduate course related to methodology, but these are all optional. The situation at Cornell is similar.

It is not known for certain whether some considerations for security, reliability, and methodology are covered in the context of other courses. For example, George Mason offers INFS 622, "Information Systems Analysis and Design", which includes "...system design practices, and management criteria in the design of large-scale information management and decision support systems." However, it is difficult to believe that security, reliability, and methodology would be covered in any depth if students are not first taught the fundamentals of those complex subjects in focused courses.

In dismay, I sought to find some schools that did better. I was not very successful at this. For example, Princeton has three undergraduate computer science tracks: Theoretical, Systems, and Applications. Only the Applications track requires course 432, "Information Security".

## The Corporate Environment

The corporate computing environment is just as problematic. Some companies have their act together, but they are few and far between. For the vast majority of US businesses, the consequences of their slackness are manageable. However, those that should be highly concerned are Fortune 500 companies that represent high-value targets, that have reputations of confidence to protect, or that have complex

systems. These companies cannot afford to be vulnerable. Yet, incredibly, most of them are.

I have found that most of these kinds of companies invest a lot in infrastructure security, monitoring, and management; but when it comes to the applications that they build, they are clueless. According to Caleb Sima, co-founder and CTO of SPI Dynamics, "Webmasters know nothing about [application] security. Instead, they're focused on network security." [1]

If an organization's applications are difficult to maintain or unreliable over time, the organization tends to ascribe it to the technology or the management of the groups who built it. With regard to security, they typically don't even know that they are vulnerable until something bad happens, so it is a game of chance, and blissful ignorance – except for those who are unlucky.

## Today's Software Development Tools

Most of the tools in use today by software developers are open source tools, created by other developers or developer-driven groups – not commercial tools created by companies. This is a significant change from only ten years ago. Further, because the tool landscape changes very rapidly today, *tool choices tend to be made by developers*, because managers assume that they are not up on the tools sufficiently in order to make credible decisions about which tools to use.

As a result, tool usage tends to be decided based on what developers *want* to use. The primary concern of a software developer with regard to a tool choice is that of making oneself marketable: developers prefer to choose the tools that give them useful experience and therefore the widest choice of employers. This means that tool choices are largely driven by fads, since fads spread quickly, and if one does not know the current tool or technology fad, then one is locked out of the job market by employers who are increasingly unwilling to invest in the time for new employees to learn tools from scratch. The secondary concern to developers is tool power: developers like to choose tools that give them the most capability to create interesting software quickly; i.e., developers like to be productive. The third concern is simplicity: developers like to learn tools that are easy to learn and use.

Notice the complete lack of focus on assurance. That is, developers are seldom interested in security, long-term reliability, or enterprise-level considerations. Developers tend to be interested in assurance-related tools only when it makes their job easier. For example, testing frameworks are very popular, because testing is usually required, and developers like to simplify and automate the task because, again, they want to make their job easier.

In contrast, a software project manager will likely have some interest in long-term reliability, security, maintainability, and manageability. Project managers are highly interested in productivity, so in that area their goals coincide with the goals of developers; but in all other areas, their goals do not coincide.

For this reason, it is important that managers have the final say about development tools and not rely solely on the advice or preferences of their developers.

Assurance-related tools exist, but programmers lack the educational preparation (let alone the inclination) to understand and use them. For example, the Solo [2] security analysis tools are costly, so developers cannot learn them on their own, and to use them, one must have an a-priori understanding of security, which we have already seen that they do not because it is not part of their education.

The Solo tools are primarily focused on finding flaws in a software implementation. There are other tools that focus more on the design phase, in order to prevent design flaws from being introduced in the first place. Tools of this genre include Model-Driven Architecture (MDA) tools [see e.g., the tools discussed on [codegeneration.net](http://codegeneration.net)], architecture tools such as Acme and ArchJava [3], and formal verification tools such as the SRI Prototype Verification System (PVS) [4]. Finally, flaws can be found through execution testing. This is the most commonly applied approach to finding flaws, and tools of this kind are abundant. Finding flaws through testing or through static analysis of an implementation are *a-posteriori* approaches; that is, verification is performed after the implementation has been created. These are preferred by programmers because little knowledge is required besides programming skill. In contrast, in order to use tools that work during the design phase, one must have other skills.

## What Must Educational Institutions Do?

In today's undergraduate computer science programs, too much attention is paid to traditional topics such as algorithms, data structures, and objects. Undergraduate curriculums need to be updated to reflect today's software challenges, which are primarily methodological. Further, students should not be able to opt out of courses that are important for establishing the foundation to be able to meet these challenges.

In particular, undergraduate degree programs should *require* courses that cover these topics, in a manner that emphasizes their importance for proper software engineering:

1. *Security fundamentals.*
2. *Reliable software engineering.*
3. Program *correctness assessment* concepts.
4. *Application-level security* concepts.
5. Software development *methodologies.*
6. At least one project related to software security or reliability.
7. An *integration of reliability and security* concepts into other kinds of software engineering courses.
8. An introduction to *specialized languages and tools* for the design of secure and reliable software, and for architecture specification and analysis.

Some of these subjects might have prerequisites. For example, assessment of program correctness might require a course in predicate calculus. That does not mean that practitioners will necessarily use predicate calculus to build software when they enter the job market, but it gives them the educational foundation with which they can understand tools that employ such techniques, so that they will not be intimidated by such tools.

Is this too much to ask? I do not think so. After all, software engineers are professionals – right? They should be expected to know these things – right? Hopefully we are not training a generation of mere

hackers.

Unfortunately, the media-reported culture of software development glamorizes those who are the least disciplined. Reports of renegade 17-year-old programmers, creating programs that threaten to take down the entertainment industry, are more newsworthy than reports about boring reliable software development. But reports about expensive and embarrassing corporate computing failures are also newsworthy – and those are a direct result of the current lack of training and discipline.

## **What Must Companies Do?**

As employers, companies play a central role in the current state of affairs, and must play a role in the solution. Let's discuss what the solution entails from an employer's perspective.

### **Demand Training, and Be Willing to Train**

As an employer, demand that your programmers are current in the topics of reliability, security, and methodology. Even if they have a PhD or many years of experience, scrutinize their training. Experience does not substitute for the right training: 20 years of poor practices is potentially no better than no experience at all.

Even if you have to pay for your programmers to take courses, demand that they do so, or that they have proven credentials in assurance-related areas. Do not accept the poor state of affairs with regard to education.

Be willing to hire people who have training in the fundamentals, but not necessarily in the latest fad technology. This is the only way to break the cycle of programmers seeking out fads for the sake of job security. Companies will find to their surprise that it actually takes very little time for a developer to come up to speed with a new tool or technology, if they have the educational foundation to understand the tools and use them properly. It is also much more desirable to have on board someone who will quickly learn and use a new properly because they understand the underlying concepts, than to have someone who has lots of hands-on experience but no theoretical foundation and who will therefore use all tools improperly and quickly create an elaborate mess. Do not tell your recruiting manager or vendor that you need "two people with ABC technology experience". Instead, tell them that you need "two people who have the background to be able to learn and properly use ABC technology." This requires some extra effort, but it is worth it.

### **Demand High-Assurance Processes**

Institute processes that ensure that what you build will not have problems. Do not succumb to the popular notion that good software is built by renegades. This may often be the case in the media, but you certainly cannot count on hiring talented renegades as a strategy for your business application group. To make reliable software development a repeatable process, you need to have a *system*, and that system must have assurance built into it. This means that you, as a business manager, must play a role in the selection of *how* software is developed. For this you might need help from experts in software development methodologies for high-assurance systems.

Do not allow your developers to choose their own tools without oversight. Their preferences should be heard, but in the end choose software development tools based on their efficacy for maintainability, reliability, repeatability, and security – as well as productivity. Do not succumb to using tools based on popularity, except for the consideration that there needs to be support for the tools in some form, and there need to be developers who know – or are willing to learn – how to use them properly.

## **Demand Independent Validation**

Insist on independent review of what you build. This includes security audits, quality audits, and certification for mission-critical systems. The only way that you know for sure that your process is producing high-assurance software on a continuing basis is to test the process by measuring what is being produced. This gives you a chance to tweak the process, based on weaknesses that are discovered through independent review.

# **What Kinds of Tools Do We Need?**

The right tools will not be created until developers understand the need for them and are driven to use them because their job requires it. Thus, we cannot solve the tool problem until we solve the educational and employer expectation problems.

If the educational and employer expectation problems are solved, the tool problem will solve itself. However, it is useful to consider the enormous gap that exists with regard to tools for building high-assurance software. Rather than merely a plethora of coding tools, we also need tools that provide for

- Pattern-conformance checking.
- Design rule expression, and checking.
- Ways to embed best practices into a design and trace it through implementation.
- Ways to define reusable patterns, based on semantics rather than structure, that can be traced through design and implementation.

These tools need to be agile, because today's business environment increasingly demands agility. They need to be simple, to make it possible for developers to learn them quickly; but they need to be based on a sound theoretical foundation and not merely hacker tools. A-posteriori tools need to be complemented by design-centric tools, because finding all problems at the implementation stage is not very agile.

## **Summary**

*It is no wonder* that we have problems with the security and reliability of our computing systems. There needs to be a call to action to address this at all levels. Educational institutions should add an assurance focus to their standard undergraduate computer science programs and address software engineering from a methodological perspective – not just from the traditional data structures and algorithms

perspective. Companies should demand more of their employees who profess to be software engineers. It is irresponsible to build information systems that hold the data of users and not do so in a manner that justifies the trust of those users. Companies should also be willing to invest in their employees, even with the risk that they will move on; otherwise, there is no chance of building the required skills.

To solve the MUTE problem will take time. There is no silver bullet. Technology will not solve it alone. Indeed, the rapid pace of technology change is part of the problem, because new technologies are keeping programmers from getting good at using the technologies. Any development tool can be used properly or improperly, and to use tools properly requires an understanding of fundamental concepts – not merely the APIs of the latest technology. That understanding of fundamentals can be provided only through education and through a willingness of companies to insist that their employees know those concepts.

## References

- 1 "App Security Still Misunderstood", eWeek, July 30, 2004.
- 2 Secure Software, <http://www.securesoftware.com/products/solo.html>
- 3 See <http://archjava.org>.
- 4 See <http://pvs.csl.sri.com/>.