

# Developer Training: Recognizing the Problems and Closing the Gaps

Mike Lyman, Cigital

**Abstract.** Problems with the way we have historically trained developers, and continue to do so, gets in the way of learning to do secure development. The same errors are repeated over and over again. Without learning the fundamentals, developers can get software to work but may lack the knowledge required to understand what is happening when it breaks. Since the goal of an attacker is to get software to break and do things the developer did not intend, that lack of understanding can become a dangerous blind spot.

## 1. Introduction

If the aviation industry operated like the software industry, the new Boeing 787 Dreamliner would have to crash several times before Boeing could learn the lessons we learned with the Comet back in the 1950s. The de Havilland Comet was the first production commercial jetliner and it suffered a series of crashes early in its service life. During the investigations into the crashes, it was discovered that the hull experienced metal fatigue as the cabin was repeatedly and rapidly pressurized and depressurized and encountered rapid temperature change while changing altitude. In some areas of the structure there were special stress points that experienced more problems than others. One of these special stress points existed in the corners of the square-cornered windows on the original versions of the Comet. The square corners caused levels of stress two to three times greater than the rest of the fuselage. The metal would fail after a number of flight cycles with one of the crashes taking place in as few as 900 flights.

Thank goodness the aviation industry doesn't operate like the software industry and the lessons learned are passed on for all. We now design airliners with an understanding of metal fatigue and the windows on airliners are now designed to have rounded corners. The Dreamliner will not suffer from problems the industry learned to fix with the Comet decades ago.

Sadly, the software industry continues to use square-cornered windows. Too many cannot seem to learn the lessons of failure others have already learned and repeat the same errors over and over again. There is clearly something wrong with the way we train developers.

## 2. Recognizing the Problems

One of the first things we need to realize is counterintuitive. We have to start by recognizing that we have to overcome our training. Problems with the way we have historically trained developers, and continue to do so, get in the way of learning to do secure development. Before we can address our gaps, we have to address our problems.

## Lack of Formal Training

There is an old joke that tells us to remember that fifty percent of all doctors graduated in the bottom half of their class. The software industry suffers the same reality with the added burden that many of our developers have no formal, college level development training. Many of our developers came at the field from other disciplines and discovered they have had the skills that it takes to write code and make software work. Not only did half of our developers graduate at the bottom of their class, many never even took the class!

The fact that so many of us are self-taught means that many of us lack the fundamentals underpinning the way our software and computers work. These fundamentals help developers understand things that are going on under the hood when we talk about issues like buffer overflows or integer overflows. Without those fundamentals, developers can still get software to work but may lack the knowledge required to understand what is happening when it breaks. Since the goal of an attacker is to get software to break and do things the developer did not intend, that lack of understanding can become a dangerous blind spot.

## Bad Habits from Instructors

Even when we do have formal instruction it can cause us problems. When I was a computer science student back in the 1980s, I remember one of my professors telling us to "forget that extra stuff, just concentrate on the lesson." That "extra stuff" included things like error checking and limiting user input that my partner and I were adding into our program because we had already accomplished the lesson and had time to do the "extra stuff." Translated, he told us to just get it to work and move on. Lesson learned.

Current computer science majors tell me their instructors are still telling them to do this. In the secure code reviews I have done over the last eight years, I see way too much of the "just get it to work and move on" in the code I have reviewed. While not an intended lesson, it is a lesson way too many developers learned and took to heart. Get it to work and move on.

Another bad habit can best be summarized by a conversation with a friend who was a computer science professor for a major university when we bumped into each other at a security conference. He said he once had a chance to see the code from a major commercial product and that it looked like the "junk" his students wrote. I pointed out his students probably did write it. It was a humorous moment but like so much humor, there is a painful truth behind it. How are developers supposed to move on from the "junk" we write as college students to being able to write quality code?

## Narrow Focus of Lessons

To be fair though, these are issues with the nature of instruction. Lessons tend to be narrowly focused because lessons with too many topics overwhelm students. Early on in their development training, students lack the foundations to understand the bigger picture so it is hard to get them to understand the lesson in a larger context. Additional topics make grading harder for the instructors; encouraging students to reach beyond the narrow lesson just adds to the instructor's work load. The added grading time may increase the chance that the instructors begin to just see if it runs and then move on which reinforces the unfortunate "get it to work and move on" lesson.

“ The fundamentals remain the same but the technology is moving so fast that books have to be based on pre-release versions in order to be available when the technology is released. Even if we use the latest materials, they are often out of date when we use them. ”

### Do the Professor and Trainers Even Know About Secure Development?

Asking professors to look beyond the narrow lessons and to include the security aspects of things assumes they understand the security implications themselves. This is a big assumption. Many of us were never taught the security side of development as we learned to develop software; it is a stretch to then expect us to be able to now turn around and teach it. It does little good to complain that instructors are not teaching secure development if we do not first train the trainers.

### Out-of-date Training

Another major problem we have to overcome is our training is usually out of date. The fundamentals remain the same but the technology is moving so fast that books have to be based on pre-release versions in order to be available when the technology is released. Even if we use the latest materials, they are often out of date when we use them. By the time a lot of developers use the training, it is even more out of date than it was when they learned it. The technology stacks we are using and the frameworks we build upon are changing so fast that it is difficult to keep up, especially when you are busy working.

### Learn it Quickly and Then Use It

Because most developers are busy working, there is a huge incentive to learn just enough to get something working and then move on; that old habit we learned from our professors when we got started. Learn a new language fast and start coding. The updated frameworks have new features developers need and they need them fast because the next release is just around the corner. Follow the examples showing how to use the new features, get it working and move on to the next thing.

### Insecure and Incomplete Examples

The problem with this approach is our examples have problems. Like all lessons, they tend to be narrowly focused to teach what is needed to be taught so they often show stripped down code that leaves off a lot of things we need for quality code. Unfortunately, those stripped down examples find their way into production code.

Beyond the narrow lessons, all too often the examples showed us how to do things exactly wrong.

One of the most glaring examples is how many developers learned to do database driven applications. The books and articles first showed us to connect to databases using connection strings like these:

```
strConnection = "SERVER=db.example.com; DRIVER={SQL Server}; DATABASE=northwind; uid=sa; pwd=;"
```

Connect to the database server with the sa account (the system admin account) and use a blank password. Why? All too often the answer from a developer was "I don't know. But the book says do it that way so we're going to do it that way." No least privilege and a very weak password.

We were also shown to do database queries exactly wrong. We were taught to create database queries by creating a stub of a query and using string concatenation to insert the dynamic values we needed to tailor the query for the specific need at runtime. Examples looked like this:

```
mysql = "SELECT * FROM tblBooks WHERE title like '%" & txtUserInput & "%'";
```

Most often, this dynamic content came from the user without any examples of input validation. This is exactly how to create a SQL Injection attack and it is what we were taught to do by the examples we were following. Combine these examples with the example connection strings that the books showed us to use and the attacker could do anything they wanted to in the database, the other databases on the server or to the server itself. Thank you insecure examples.

### 3. Overcoming the Problems

Before we can begin to address the gaps in our development training that lead to security issues, we have to address the problems our historical approaches to training have caused.

#### Stop Teaching Bad Habits

One of the first things we need to do to overcome these problems is to stop teaching the students bad habits. When students move beyond the lesson and start adding in "extras" like error checking and limited user input, do not discourage them. This may mean instructors have to look deeper to grade the actual lesson but doing so saves employers from having to overcome the "get it to work and move on" lesson so many students have taken to heart. It would be beneficial to actually encourage the students to do more than just get it to work, especially when they have moved beyond the basics and on to more advanced lessons.

We also need to stop letting students write junk code. They need to use meaningful variable names, properly document their code and follow standards. The code needs to be easy to read. Many developers do not learn these lessons until they have to maintain junk code and feel that pain. Early lessons that exist only to force them to do maintenance on badly written code and feel that pain will help encourage them to write better code. As an added benefit, the better code will be easier to grade for the instructors.

“ We have been doing it wrong for a long time. The tragedy isn't that we cannot seem to learn the lessons of history but that we have learned the lessons and all too often we have failed to pass them on. ”

### Remind Students There is More than the Narrow Lesson

While there may be no way to get away from narrowly focused lessons, we should constantly be reminding students that the lessons do have a narrow focus and in the real world there is more you must do. Instructors should show the lessons applied in a larger context. Periodic summary projects where several narrow lessons are wrapped into a larger project specifically focused on the bigger pictures will reinforce the concept. Part of this bigger picture must include the security implication of what the students are creating.

### Train the Trainers

Unfortunately, to inject the security picture into our lessons, the instructors have to know that security picture. Like so many of us, they probably never learned the security implications of what they are doing. It is critical that we train the trainers so they know secure development. If they do not start to teach new developers how to write secure code from the beginning, we will always be playing catch up. We will have to teach them not to use square-cornered windows while they are on the job. We need to teach the teachers so they can properly teach their students.

### Purge the Bad Examples

We must purge bad examples from our lessons. We learn from examples and using insecure and badly written examples creates bad habits that have to be unlearned later. The development training publishing houses are doing a much better job of this today than they used to, but they need to remain vigilant. They need security focused reviewers going over code examples just like we need security focused reviews of real code. When we are the instructors we need to be careful about the examples we use. We need to go over our lesson plans, especially old ones, and purge the insecure examples.

### Students Have to Do Their Part

All of this cannot just fall on the trainers and producers of training materials. Students, especially professionals learning new technologies, must do their part. They have to remain aware that lessons are deliberately narrow and remember there is a bigger picture. We have to remember that while the instructors are only grading the lesson they taught, there are a lot of other important things we have to do with our code. We have to be

aware that examples we follow are also narrowly focused and be diligent about making sure we learn what else we need to be taking care of in our code. We have to be aware that the examples we follow may not be the most secure way of doing things, especially when the examples are old and out of date. We have to be active students when it comes to security.

### 4. Closing the Gaps – Learning from History and Today

Once we begin to overcome our training, we can begin to close the gaps that lead to security problems.

The most fundamental gap is that developer training ignores a wealth of history of software failures. We have suffered from decades of buffer overflows and race conditions, input validation failures and injection attacks. We have created software with unnecessary features that cause security issues enabled by default rather than being an optional feature users have to enable. We have been doing it wrong for a long time. The tragedy isn't that we cannot seem to learn the lessons of history but that we have learned the lessons and all too often we have failed to pass them on. The bigger tragedy is those lessons, when we do pass them on, are reserved for “secure development” classes instead of being incorporated into development classes in general.

It would not be difficult. Inject the lessons from failure right along with the actual lessons. We are all used to sidebars in our books. While the main lessons teach new developers how to write code without the issues that cause us problems, the sidebars can tell the stories of how we once did it wrong. They can tell the stories of the money it cost companies or the lives lost. They can link to the historic issues in the Common Weakness Enumerations and Common Vulnerabilities and Exposures databases that MITRE maintains. Show the developers that bad code has consequences beyond a bad grade in class. Rather than teach students to do input and then teach validation at a much later date, teach them to accept limited input that is immediately validated and move on to how to accept less restrictive input at a later date. When teaching numeric types, also teach how computer treat numbers differently than humans do. Have lessons early on that deliberately show the impact of numeric overflows. When having to pass commands to other command processors, teach students to use parameterization mechanisms so the receiving end can easily tell what the command the developer specified was and what is input into that command from other sources. Show them what happens when they do not do it that way. And all along, have sidebars talking about real examples of what happened when we got it wrong in the past.

We've got to get past teaching developers this is the way to do something and later teaching them the secure way to do it. Let's just teach them the right way from the beginning and never let them learn the insecure way.

Even when we learn the right way up front, developers will still have to learn about software failures that occur today. These may be repeats of the lessons from history or they may be new lessons as attackers continue to become more creative in attacking our software. Often, this can come from continuing

## ABOUT THE AUTHOR



**Mike Lyman** is a senior security consultant at Cigital. His areas of expertise include secure code review, vulnerability assessments and training developers in secure development. Mike spent 12 years with SAIC helping create their software assurance offering for DoD customers at Redstone Arsenal, AL; pioneering most of the processes and procedures used by the practice. He has been a CSSLP since 2008 and a CISSP since 2002.

**21351 Ridgetop Circle, Suite 400**

**Dulles, VA 20166-6503**

**Phone: (800) 824-0022**

**Fax: (703) 404-9295**

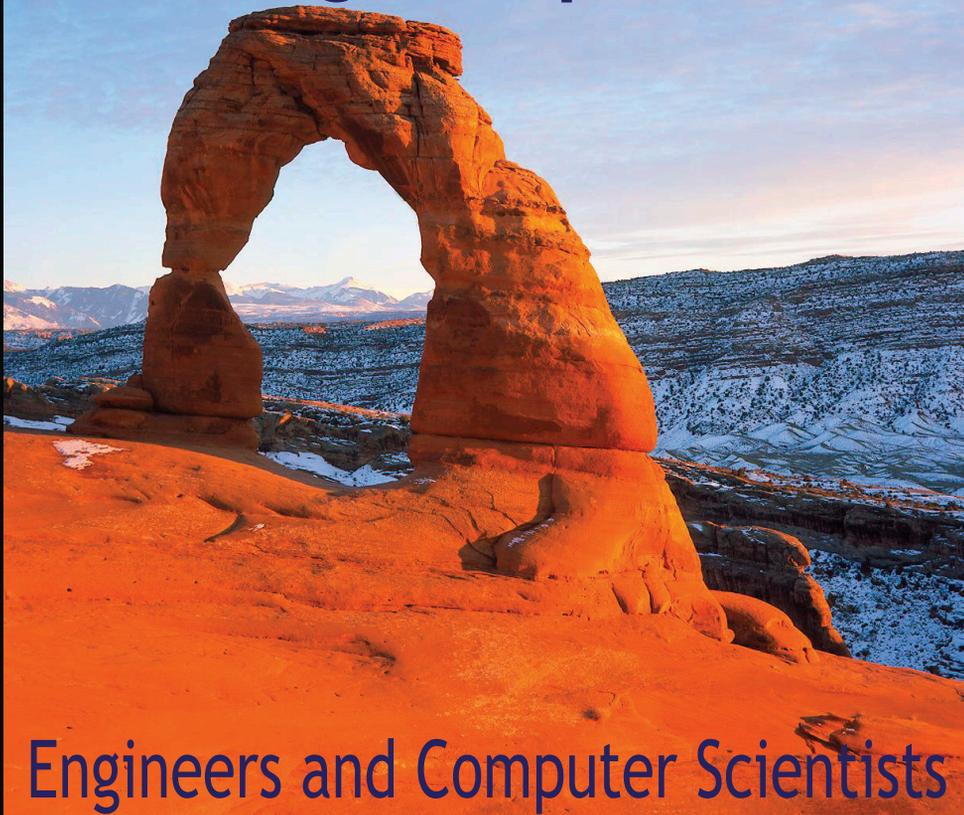
**E-mail: [communications@cigital.com](mailto:communications@cigital.com)**

secure development training as we continue our education. Some of this needs to come during our code reviews as we cover the problems in our code. We need to deploy lightweight static analysis tools to the developer workstations that can catch mistakes as we create them, similar to the way spell checkers work today, and then the tools teach them the right way to do things. The instructors, code reviewers and tools must stay up to date with the latest trends.

Within organizations, trends in their own code should be shared organization wide, especially when there is a significant failure. Imagine the airline industry if the lesson learned from the Comet had not been shared. If organizations are brave enough, they can share the lessons learned with those outside the organization similar to the way Microsoft's SDL blog did on occasion after a patch Tuesday. Share the lesson learned. The wider the lessons are shared, the better for all of us.

Other engineering disciplines have successfully merged learning from failures into their basic education and continuing to learn from new failures in their continuing education. They no longer make the same mistakes over and over like the software industry does. Because of this, we no longer have to worry about boarding a new airliner with square windows feeding metal fatigue problems in the fuselage. Wouldn't it be nice if the developers of our shiny new software had also learned the lessons of history and did not recreate problems we learned to avoid long ago?

# Hiring Expertise



## Engineers and Computer Scientists

**T**he Software Maintenance Group at Hill Air Force Base is recruiting **civilians** (U.S. Citizenship Required). Benefits include paid vacation, health care plans, matching retirement fund, tuition assistance, paid time for fitness activities, and workforce stability with 150 positions added each year over the last 5 years.

**Become part of the best and brightest!**

Hill Air Force Base is located close to the Wasatch and Uinta mountains with skiing, hiking, biking, boating, golfing, and many other recreational activities just a few minutes away.



**Send resumes to:**

**309SMXG.Recruiting@us.af.mil**  
or call (801) 777-9828



[www.facebook.com/](https://www.facebook.com/309SoftwareMaintenanceGroup)

309SoftwareMaintenanceGroup