Worst Practices in Developing Secure Software

I define "Best Practices" as practices validated by experience and common sense. Often the "common sense" component is entirely skipped and the "Best Practices Mantra" is used as an excuse for not thinking. That includes not thinking about what "Best Practices" actually means. A quick google reveals lots of "Best Practice" links and "Best Practice Institutes," but none of them bother to define "Best Practices." No one bothers to define the Bible or Koran either, but I would argue that they do not need to be defined, and are certainly not fads. The jury is out on "Best Practices" . . .

I was at a new client's facility yesterday and when I asked about certain security practices and configurations, I often got the blanket statement "Best Practices" as an answer. It was clear in several cases that these alleged "Best Practices" either did NOT apply or were INCORRECTLY implemented. In both cases no one had used their common sense and had instead chosen to place blind faith in someone's "Best Practices."

A major component of success involves avoiding making any major mistakes. Instead of focusing exclusively on implementing "Best Practices," while useful if done correctly, I suggest also avoiding "Worst Practices." You can do almost everything perfectly, but if you get one thing horribly wrong you can negate everything. A soldier greatly increases his chances in a firefight by doing things right, but one serious mistake and his odds of surviving plummet. Fatal flaws and mistakes are exactly that – FATAL!

Many problems with computers today come insecure software – software that isn't developed with security in mind. This includes software that is badly designed, badly coded, and that has serious operational issues – like insecure and buggy installation techniques and default accounts and passwords intact.

If everyone avoiding the following "Worst Practices," computers and the Internet would be a safer and more productive place.

Assuming that only "important" software needs to be secure.

All programs and services need to be secure. Even a simple game or utility could be compromised, contain a Trojan or otherwise harbor malicious code, and lead to your entire network being compromised.

Prototype code can be compromised as well and often ends up incorporated into production applications. Test code can also be compromised. As much as possible, prototype and test code need to be well written and secure. Of course it would be naive to state that prototype and test code can always receive the same amount of attention to detail as production code and can always be as well written, but they should certainly not be "garbage code."

Emphasizing hitting deadlines ahead of writing "Good Code."

Deadlines are important, but not at the expense of writing decent code! Can you image an engineering project in the physical world taking the equivalent attitude? "That bridge will open August 1st no matter what. We'll let the bridge users uncover any unresolved problems and patch them later."

Of course it's easy to preach when in the pulpit, and harder when in the foxhole. Management, customers, and others might be screaming for the latest release. In extreme cases getting a software release out soon, any release, might determine whether the company still exists in a few weeks or not . . .

Bad code goes hand in hand with increased bugs, security problems, and long term expenses. In the long term, good code pays.

Having IT make all risk management decisions.

Every business decision involves risk. Many important IT decisions are important business decisions and can involve significant risk.
Some IT decisions involve enough risk that executive management should be involved in the decision making process.
For example, how much risk and which risks are acceptable in protecting the entire customer database which may include credit card and other sensitive information? This is NOT a decision that IT should make alone! It is a risk management decision that requires input from upper level management.

Not considering security during the entire application lifecycle.

Security should be considered during the entire application lifecycle. Security must be part of the system design based on the product security goals, attention must be paid to security during implementation, and operational issues including installation are critical as well.

Adding appropriate security later is difficult if not impossible because the design may be fundamentally insecure. Adding security can also cause changes to features and/or application interfaces affecting backwards compatibility.

Adding security is more time consuming and resource intensive than doing it "right" from the beginning, hence it is less likely to be done well or given due diligence.

Assuming the software won't be attacked.

Most software is attacked! Don't make the following common and usually false assumptions:
–The users are friendly
–Input will not be malicious
–The environment is hospitable
–The firewall will shield the application from hostile users and attacks

Not doing any security testing.

Security testing is *much* different than functional testing and both are important. Instead of examining a system's response under fairly normal circumstances, security testing involves probing the system looking for weaknesses much like an attacker would.

Security test plans are critical. Just haphazardly "testing things," while not entirely useless, is very far from ideal. The Test Plan, including security testing, should be an integral part of the systems development.

Testing is NOT a replacement for good design and implementation. It can discover vulnerabilities, but cannot prove that no vulnerabilities exist.

Not planning for failure

Complex systems can and do fail. Both partial and complete failures need to be planned for. Software should always fail to a secure mode, and when in failure mode, deny access by default. If the entire system fails, any secure data should be unavailable!

When failure occurs, no data should be disclosed that wouldn't normally be available, and as little information as possible should be disclosed.

For example, if a login fails, it is far preferable to report that the login failed than to specify "invalid password" or "no such account." If a login fails, it should reveal no information other than failure (if even that).

I worked on one system where the results of a successful and unsuccessful login were visually the same – the user didn't even know their login failed until they tried to do something.

In contrast, I recently was authorized by a client to login to their Blogger account to add Google Ads and make a few other changes to their blog. They gave me an incorrect account name and password, and when I tried to login I got a message that said "non-existent account." I now knew the account name was wrong, and I tried a couple of "obvious" account names such as the company name, etc. My second guess was correct and I got a different error message, "incorrect password." The password was easy to guess too – it was my client's dog's name!

I was authorized to access his account, but even if I hadn't been, I could have "guessed my way in." If Blogger didn't differentiate between incorrect passwords and non-existent accounts, it would be more secure and I probably would have given up quickly and waited for my client to give me the correct login information.

Should a significant failure occur to a critical system, e.g. a defacement of the organizations web server or a inability of a server used for ecommerce to authorize credit card purchases, there *should* be a security policy in place that specifies contingency plans. For example, should the server be taken off line? Should it report an "unavailable – try again" message? Should it be left live and fixed as quickly as possible?

Counting on "Security through Obscurity"

Security through Obscurity is the notion that hidden vulnerabilities will not be discovered. It can be used as a part of a Defense in Depth strategy, but should never be depended on alone. Secrets are hard to keep!

For example, not releasing source does not guarantee that any secrets in the binaries will remain secret! Binary code can be reverse engineered, disassembled, or decompiled.

Although full disclosure of cryptography code is perhaps somewhat controversial, most IT professionals believe that it leads to more security as more people can easily examine the code for vulnerabilities.

I actually LOVE security through obscurity as part of a defense in depth strategy! There is nothing wrong with keeping secrets! There is no reason to make life easy for hackers. For example, why advertise the OS, version number and patch level you run? Why let anyone know anything about your firewall? Why let whether you run Apache or IIS or some other webserver be public knowledge? There is typically no reason to publicize any of these, but don't count on them remaining secret!

Disallowing bad input instead of only allowing good input.

A lot of previously discovered vulnerabilities rely on malicious input.

Input should always be validated. For example, if the program expects an 20 digit number confirm a 20 digit number is input, if an address is expected make sure the input is in the form of an address. Always check the size of the input!

Invalid input should not be rejected – instead only valid input should be allowed. There are too many possible types of invalid input and something may be missed otherwise. For example input may be in encoded in hex, Unicode, or some other unexpected format.

Applications should have a trust boundary defined around then, with a small number of entry points through the trust boundary. All input should pass through and be validated by one of those entry points. This includes not only input from users and other applications, but also config files, environment variables, etc.

Software that is not secure by default

By default, a system should be secure when installed. All resources should have adequate protections be default. Rarely used features shouldn't be installed by default as they increase the attack possibilities.

This is all easy to say, but more difficult to achieve. What does reasonably secure mean? In the past there has been a trend for default installations to have minimal security configured, for example in Windows and Unix/Linux operating systems. The more

secure, the more difficult is it for users and (inexperienced) administrators to get work done. This may result in more difficulty in getting a system up and running initially and in training new users. It may result in more calls to help desks etc.

What reasonably secure means depends on the system and environment. However it is typically MUCH easier to loosen security later than to tighten it.

Rolling your own cryptography

Cryptography is its own very complex and difficult discipline. Programmers are NOT cryptographers and should not be developing OR implementing cryptographic algorithms with VERY few exceptions!

Something proven and commercial strength should be used. For example Microsoft Windows (and other operating systems) include cryptography services that can be used by applications.

The classic bad example of "rolling your own cryptography" is the DVD Content Scrambling System (CSS).

It was certainly NOT designed by cryptographers and a number of weaknesses exist. Decryption Code for CSS, DeCSS, was written pretty quickly and posted on the Internet by a Norwegian teenager Jon Johansen.

Proven commercial strength encryption should have been used! Then again, maybe the programmers thought it was a stupid idea and wanted it to fail?

About the Author

Ted Demopoulos' first significant exposure to computers was in 1977 when he had unlimited access to his high school's PDP-11, and hacked at it incessantly. He consequently almost flunked out but learned he liked playing with computers a lot.

Ted's first business ventures began in college and have been continuous ever since. In 1990, he founded Demopoulos Associates, which concentrates on Information Security and ITY Entrepreneurial Issues. Ted holds several industry certifications but is reticent to append initials to his name to his already long name. He also serves as a Senior Member of The Institute for Advanced Professional Services and Board Director of The Security Training Institute.

Ted blogs at http://www.thetedrap.com/

References:

Books:

Building Secure Software: How to Avoid Security Problems the Right Way,
by John Viega and Gary McGraw, Addison-Wesley,
ISBN: 020172152X

Writing Secure Code, Second Edition
by Michael Howard and David C. LeBlanc, Microsoft Press,

ISBN: 0735617228

Secure Coding: Principles and Practices
by Mark G. Graff, Kenneth R. Van Wyk, Oreilly, **ISBN:** 0596002424