

(IEEE SECURITY & PRIVACY EXCLUSIVE CONTENT)

BASIC TRAINING

How to Think about Security

James A. Whittaker and Richard Ford

Florida Institute of Technology

Learning how to think about security means adopting a different mindset than we've had in the past. As a community, software developers have been thinking too much like "good guys" and thus ended up developing insecure software because they failed to predict attack scenarios. The only way to effectively develop good security in software is to learn to think like the "bad guys." Thinking like the adversary helps us to better identify and mitigate threats.

Increased adoption of threat-modeling techniques has led to significant progress in this area, but such techniques are just the tip of the iceberg. Threat modeling, though useful, is only as good as your ability to understand the real threat. It's not enough to think about and nicely document single-attack scenarios. We need to view the system on a much wider basis, and that's impossible if you don't know how to really think about security.



MAR./APR. 2006

Robbing the bank

Consider the example of insecurity represented by bank robbery. (Andre Dos Santos describes this attack example at www.cs.ucsb.edu/~kemm/courses/cs177/wspec00.pdf.) A far less dangerous crime than it used to be, bank robbing no longer requires daring daytime raids by masked, shotgun-toting thieves. It's safer, more expedient, and almost certainly more comfortable to simply hack your way to wealth from the comfort of your own computer—or, better yet, someone else's.

After choosing a target bank, high-tech robbers can "case the joint"—gather information that will assist the actual theft—by simply aiming a browser at the bank's Web site and clicking on the "establish a new account" link. Creating an account (preferably under a false identity) lets us play around with the target site and gives us access to the pages that legitimate customers use to learn how the site is put together. It's the equivalent of walking around a brick-and-mortar bank to assess the vault placement, security camera positions, and security guard routines. Better yet, we're not stuck with a particular physical location. Casing multiple banks out in the real world is time-consuming and puts your face on camera. Casing banks online is pretty easy, on the other hand, and if bank A doesn't look promising, bank B is just a few clicks away.

We might get lucky and find obvious weaknesses at our target site, but even if we assume the developers did their job and didn't leave us any SQL-injection or XSS vulnerabilities, we're still likely to get some very useful information. For example, the bank gives us an account number and informs us that we need to set up our preferred authentication method. In addition to personal identification numbers (PINs), modern systems often add extra information, such as a randomly selected personal question on predetermined topics such as your pet's name.

Bank account numbers are interesting. Despite the number's length, the variability isn't as large as it might first seem. Historically, bank account numbers have been sequential, or included an underlying sequential numbering plus a check digit (to detect keying errors by operators). Thus, it's not beyond imagining that we could guess other account numbers once we've received one (or several, given the time to sign up for multiple accounts).

Of course, even if we can't guess account numbers, it's probably not inconceivable to brute-force some, especially if each account is protected only by one of 10,000 different four-digit PINs. Given that this is a very small number for computers, we might decide to try a brute-force attack on the login page: we start by typing a random account number and guessing 0000 for the PIN. When we get an incorrect login we try again (which is what brute-forcing is all about) with 0001. Same result. When we use 0002 as the third try, the Web site's security kicks in and locks the account—a standard defense to protect online accounts, usually locking them for 24 hours or until the affected user calls the bank to unlock the account.

Sounds pretty good, eh? That means no accounts were compromised, so the security solution works—but only if we're thinking like a good guy. If we think like a bad guy, several attack opportunities are clear.

To begin, attackers can abuse this three-strikes-and-you're-out feature to cause complete chaos. Imagine, for example, the instant call-center overload that would ensue if an attacker cycled through all the accounts, locking them one after another. Writing such a program is trivial with any of today's programming languages, and when it's finished, every bank customer is locked out. Although they haven't walked away with cash, the attackers have managed to seriously inconvenience the bank's legitimate clients and perhaps caused the bank to lose customers. Certainly, the attack will cause the bank to change its approach to security, and if the site's architecture can't handle the change, costly rework will be needed. As a general rule, organizations tend to shy away from adding such additional security as it becomes more difficult or more expensive to manage. If the warning light comes on too often, experience tells us that users will ignore it or, worse yet, turn it off completely.

Furthermore, the three-strikes-and-you're-out rule gives the bank and its users a false sense of security. From the perspective of a single account, the protection is quite good—if someone is trying to break into a single account. However, crooks seldom think that way. After all, the money is all the same, regardless of which account it comes from. Instead, we could pick two common PINs, say 1234 and 1111, and try all the account numbers we could with them. The defensive mechanism wouldn't trigger because we'd never guess three consecutive times on a specific account number. Instead, the result (over sufficient time) would be access to every account the bank owned. The attacker could even insert random delays to mask the attack's automated nature. Game over, the vault has been cleaned.

What went wrong here? Why was the bank's site so easy to penetrate? We believe the answer is that most software designers and developers think the wrong way about security.

True understanding vs. blindly securing against known attacks

Any trip to a local bookstore will provide ample proof of our fundamental point that most security teaching is about checklists rather than understanding. The reason for this is entirely pragmatic: if we're teaching about securing Windows or Linux, we can provide a checklist that ensures that a machine is robust to penetration—essentially, follow these simple steps and everything will be all right.

Of course, the problem with this technique is that it's not very advanced. For example, from the perspective of Benjamin Bloom's taxonomy—which ranks learning from simple levels (like simple recall) to more complex and abstract (such as evaluation and synthesis)—most security classes really don't approach any of the higher learning levels at all.^{1,2} Instead, the focus is on learning what to do, with little attention to why—let alone concepts such as synthesis and material evaluation.

We believe this situation results partly from the explosive interest in security. With the field moving so quickly, teaching tends to focus on achieving short-term results to large challenges that require immediate practical solutions. However, this focus also means that fewer people really understand security. Checklists are all well and good, but they're a terrible medium for developing and distributing understanding.

Rather than individual "do this, don't do that" instruction on how to apply preventatives, a focus on deeper understanding of the issues is crucial. Security makes sense only in the context of systems. In the bank example, each individual account was protected, but the system as a whole was vulnerable.

This might seem to fly in the face of statistics that show that most attacks are well-known, and all would indeed have been well if defenders had followed best practices. However, reactive approaches aren't working. More seriously, they aren't about to start working because checklists constantly change, leaving defenders one or two steps behind attackers. Although taking reasonable precautions is important, thinking about security picks up where naïve threat modeling leaves off.

Protecting all users

Universities teach things like requirements analysis and techniques such as Unified Modeling Language to get budding developers to consider how a specific user will interact with a system. We want usable software that accomplishes the tasks that the user really needs.

All these techniques are useful, but we must realize that any single user's view of a system is necessarily limited and describes only one dimension of the application's total functionality. Developers must balance this microscopic view with a macro level view that considers all of the system's users.

In the bank robbery, this microscopic viewpoint created a system designed to protect a single account—a feature that an attacker could use not only against the rest of the user community but also to gain access to every account on the system.

Protecting a single user should never be the goal. We need to protect them all. Had this been their focus, the developers would have realized that the predictability of the account numbers and PINs allowed the hack we outlined. Putting more thought into making these harder to guess would have yielded a more secure system than the three-strikes-and-you're-out rule.

Economy of scale is another issue. Computers are good at mindless, repetitive tasks, and so brute-forcing passwords and PINs is something at which they excel. Developers should remember that attackers have such tools and defend against their use.

It's about what the software shouldn't do

As developers, we learn to specify correct behavior and then code it. As testers, we learn to develop test plans and then implement them. As such, statements like the following are common:

When the user applies input *a*, the system should produce output *b*.

A developer who sees this requirement will dutifully write code that allows input *a* to be entered and then code the functionality necessary to render output *b*. Once this function gets to the quality assurance (QA) team, an engineer will apply *a* and then ensure that output *b* is returned.

Granted the function produces the right answer, all is well, right?

Wrong. The problem is that security isn't about producing the right answer. Indeed, the fact that the right answer occurred actually masks potential security issues. QA folks are trained to perform specification-based testing and are only too happy to check test cases as "passed" when they get the right response, and then move on to the next one.

Security is about what went into producing the right answer. Specifically, the question is whether all the computation and data manipulation that took place was well-behaved or whether it opened an exploitable attack vector. Rather than looking at the final response, security-minded folks look instead at the process that generated it and ask themselves, "What might have gone wrong?" In other words, we have to change the above requirement to:

When the user applies input *a*, the system should produce output *b* and should not produce side effect *c*.

But thinking through what shouldn't happen is a much harder task. How do we enumerate every possible *c* in this requirement?

A good way to think about this issue is to consider the side effects of the functionality. In the bank example, a side effect of locking a user's account is locking all users' accounts by using the legitimate functionality against the software.

Digital rights management provides a classic example of this. Suppose you're testing a music player that encrypts the file so that it's bound to a single player. You play the music and all is fine, so you copy the file to another player and it doesn't play. Finished? No. Now you must ensure that an attacker can't easily break the music player implementation. If, for example, the player decrypts the entire file and writes it to disk (perhaps as a hidden file in a hidden directory), the unencrypted file is exposed. These are the details and side effects that should interest the security-minded developer. The correct response must occur (playing the music), but it also needs to happen without insecure side effects. Developers and testers need to stop looking at the forest and start analyzing the trees.

Never assume the world works as you think it does

Imagine being a QA professional and finding what you consider to be a bug in the system you're testing, only to have a developer pooh-pooh it with the six words most loathed by testers the world over: "No user would ever do that!"

This phrase has been said so often at software-development shops (as an excuse not to fix a bug) that many have begun to believe it. As a community, we've grown used to making assumptions about our user base, including that they won't purposefully do stupid things or abuse systems that they rely on.

We might get by with that attitude toward other aspects of quality, but not with security. Malicious users often have the same access as legitimate users. Nothing prevents them from attempting log-ons, obtaining accounts, or browsing our applications. And no rule prevents a legitimate user from changing hats to become our adversary.

The best example is in the Web domain where developers often assume that the browser will protect their servers by restricting what users can do. Yet, browsers exist on client machines and are out of our control. Proxies, scripts, and techniques such as forceful browsing (visiting pages by entering URLs directly rather than following the Web site's navigation links) are proof that browsers can't be trusted. In every system, we must understand what's expected and what's guaranteed. We can expect a browser's JavaScript implementation to conclude that $1+1=2$, but we can't guarantee it, and we certainly can't rely on it for any security-related decision.

Sure, most users are legitimate, and hacks are rare events. But it takes just one malicious user to bring a system down or exploit it in a way that ensures that it's never trusted again. The assumptions we make about our users require a little bit of paranoia. Trust can't be assumed; it must be enforced.

As if user unpredictability weren't problem enough, developers also have to tangle with the unpredictability of the environments in which their applications reside. Given the vast environmental variations that our software might encounter, we have to anticipate how an application will react in different scenarios. Expecting the unexpected isn't just about adversarial users but also about hostile environments. Just because you built the application to run on a standalone Macintosh doesn't mean that it won't be installed on a machine on an open network. Such unforeseen, and therefore untested, outcomes present substantial risk to users. The moment the application's installed on a networked computer, it becomes a target.

Security checklists are important, and all developers and system administrators should use them, taking pains to ensure that the lists are current with respect to modern threats. Yet, the truth is that keeping track of adversaries and their techniques is a full-time job that's never likely to be complete.

Checklists are not enough and, worse, they make us focus on single-point solutions and treat security as a series of bandages on top of working systems. We must instead peer into our adversaries' minds and understand their true intentions even more than we understand their techniques. This requires that we analyze systems as a whole and consider attackers' real goals in breaching our systems.

References

1. B.S. Bloom, *Taxonomy of Educational Objectives, Handbook I: The Cognitive Domain*, David McKay Co., 1956.
2. D.R. Krathwohl, B.S. Bloom, and B.M. Bertram, *Taxonomy of Educational Objectives, the Classification of Educational Goals. Handbook II:*

Affective Domain, David McKay Co., 1973.

James A. Whittaker is a professor of computer science at the Florida Institute of Technology. His research interests include security testing and software engineering. Whittaker has a PhD in computer science from the University of Tennessee. He is the author of the newly published *How to Break Web Software* (Addison-Wesley, 2005).

Richard Ford is an associate professor at the Florida Institute of Technology. His research interests include malicious mobile code, spyware detection and prevention, and security metrics. Ford has a PhD in physics from the University of Oxford, England. He is a consulting editor for *Virus Bulletin*.

Editors: [Michael Howard](#) and [James A. Whittaker](#)

Cite this article:

James A. Whittaker and Richard Ford, "How to Think about Security," *IEEE Security & Privacy*, vol. 4, no. 2, March/April 2006, pp. 68-71.



PRINT



CLOSE