SOFTWARE SECURITY SERIES

ENTERPRISE SOFTWARE SECURITY

A CONFLUENCE OF DISCIPLINES

Kenneth R. van Wyk ▪ Mark G. Graff
Dan S. Peters ▪ Diana L. Burley, Ph.D.

Enterprise Software Security

Enterprise Software Security

A Confluence of Disciplines

Kenneth R. van Wyk
Mark G. Graff
Dan S. Peters
Diana L. Burley, Ph.D.

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco

New York • Toronto • Montreal • London • Munich • Paris • Madrid

Capetown • Sydney • Tokyo • Singapore • Mexico City

Library of Congress Control Number: 2014950276

ISBN-13: 978-0-321-60411-8
ISBN-10: 0-321-60411-3

Text printed in the United States on recycled paper at RR Donnelley in Crawfordsville, IN.

First printing: December 2014

I want to dedicate this book to my wife and my parents,
for believing in me and encouraging me.
—Kenneth R. van Wyk

I want to thank my old friend KRvW once again for his
trust and patience, as well as for the
astounding insight that launched this adventure.
—Mark G. Graff

My work is dedicated to my family and to
childhood memories about my grandparents.
—Dan Peters

To my family for their unwavering support.
—Diana L. Burley, Ph.D.

Contents

Acknowledgments

From Kenneth R. van Wyk:

Thanks of course to my authoring team. We come from a pretty diverse set of backgrounds and experiences—by design—and I'm proud of what we've put together here. Together we proudly wave the flag of confluence and hope many will join in.

Likewise, our tech reviewers have been fabulous: Danny Smith (who has been on a couple of my review teams now), Bill Reynolds, Derrick Scholl, Andrew van der Stock, and Kevin Wall. Like the authors, you represent a diverse set of experiences, and we value greatly all your suggestions. I hope we've done them justice. If we haven't, it certainly isn't your fault!

I also want to send out a special thanks to my mom and dad. My dad, who as a schoolboy was captured by the sounds of those Rolls Royce Merlin and Griffin engines in their Spitfires and Hurricanes as they flew over his grade school. He later went on to be a 747 pilot at what was at the time one of the biggest and most respected airlines on the planet. He encouraged me in everything I did. Every hobby, everything. He always encouraged me to soar, whether it meant competing or simply pushing myself to the limits of my abilities. I'm forever grateful for that encouragement.

And my mom, who not once, not twice, but three times followed her husband and moved to three different continents. She was a farm girl from rural South Africa, where English was anything but the primary language. And then, she not only learned English, but became an English professor at a U.S. university. I often consider how I would react to being, say, a French professor to French students, and it gives me chills.

How could I fail, given such amazing examples! Thanks, Mom and Dad. I love you both.

From Mark G. Graff:

I want to thank my old friend KRvW once again for his trust and patience, as well as for the astounding insight that launched this adventure (maybe better: his "trust, insight, and astounding patience"). To Diana and

Dan: There would be no book without your skills and dedication; I feel rescued. To my family: Wow—look what we did! Finally, love and fond regards to fellow security practitioners/sufferers around the world.

From Dan S. Peters:

I feel proud to be part of this exquisite author group and to able to contribute my knowledge and experience to this book project. Even though I could anticipate ahead of time all the stress related to book writing, it was a very easy decision for me to take part in the project once I talked to Ken and Mark about it. The arguments were simply so convincing and aligned so well with my vision of the situation that I could not resist the temptation. Together with Diana, the four of us formed a well-rounded team, complementing each other's vision, knowledge, and passion for the subject of software security. I believe that the confluence book would not be possible based on the experiences of individual authors, and could only be born out of such collaboration.

My family, of course, deserves appreciation as well, as they often had to cope with weekend calls and late-night shifts while I was trying to move forward on an especially stubborn subject. They had to deal with my frustrations about writer's block and continued encouraging me in this endeavor through all these weeks, months, and (alas) years. Thank you!

From Diana L. Burley:

Thank you to this amazing team of authors. Ken and Mark, our synergy was apparent from the start, and I was honored to work alongside you as your vision turned into our vision and then into our reality. Dan, your ability to hone in on the technical details of every discussion sharpened us all. Our collaboration is a call to action. To the educators who will answer this call, the future begins with you. Let the message of confluence permeate your classrooms as you develop the next generation of security professionals.

To my family—my children, who inspire me every day to push beyond traditional boundaries and challenge the status quo; my parents, who instilled in me the need to disrupt accepted thought patterns by crafting arguments that blend academic rigor (from my mother, the college professor and administrator) with practical wisdom (from my father, the corporate executive); my brother, whose belief in me carried me through the late nights; and my best friend, who makes each day *always better*!

About the Authors

Kenneth R. van Wyk is a career security guy, having started with Carnegie Mellon University's CERT/CC in the late 1980s and subsequently worked for the United States Department of Defense and in several senior technologist roles in the commercial sector. He is the co-author of two popular O'Reilly and Associates books on incident response and secure coding. He now owns and runs KRvW Associates, LLC, a software security consulting and training practice in Virginia, USA.

Mark G. Graff is the CISO of NASDAQ OMX. Formerly the chief cyber-security strategist at Lawrence Livermore National Laboratory, he has appeared as an expert witness on computer security before Congress and analyzed electronic voting machine software security for the state of California. A past chairman of the International Forum of Incident Response and Security Teams (FIRST), Graff has lectured on risk analysis, the future of cyber security, and privacy before the American Academy for the Advancement of Science, the Federal Communications Commission (FCC), the Pentagon, and many U.S. national security facilities and think tanks.

Dan S. Peters has been involved with security for longer than he had first expected when he stumbled into this field out of curiosity while making a good living as a consultant and a commercial software developer. Many security disciplines are exciting to him, but mobile security has been the most intriguing topic as of late. Before working on this book, Dan repeatedly shared his passion for security in conference presentations and numerous publications.

Diana L. Burley, Ph.D., is an award-winning cyber-security workforce expert who has been honored by the U.S. Federal CIO Council and was named the CISSE 2014 Cybersecurity Educator of the Year. As a professor, researcher, and consultant on IT use and workforce development for nearly 20 years, she passionately promotes a holistic view of cyber security to influence education, policy, and practice from her home in the Washington, D.C., region.

Preface

In today's commercial enterprises, information security staffs spend years building walls around their business applications. That's good. Practitioners have known for years, however, that—for a real chance at corporate safety—the enterprise's application programmers must also build security into the business software.

Yet even the powerful combination of a sound perimeter and front-to-back application security might not suffice against the highly sophisticated attacks launched against today's networks. One surprising reason: There is all too often a cultural and physical separation between the software development staff and the information security staff in large enterprises.

This book bridges that gulf. We identify the issues that distinguish and keep the two groups apart, and suggest practical and actionable guidance as to how best to collaboratively address the security needs of the enterprise. This book will help programmers design, write, deploy, and operate better enterprise software applications. It will help network security engineers make better use of the applications' output to drive and adjust manifold security appliances, such as firewalls. But we hope it will achieve much more.

Uniquely drawing ideas from two distinct disciplines, software engineering and network security, this book is intended to point the way to a new, holistic approach to enterprise protection. We envision an integrated program of application development, security appliances, network architecture, and policies and procedures we call "confluence."

It's not just about the perimeter anymore, or even safe software. Recent developments such as the so-called "advanced persistent threat" have breached those barriers. But in this book we show how businesses can move forward by building software that actively contributes to the intrusion detection and response processes. (Today, most extant enterprise software still provides audit logs or event logs of security exceptions, and little more.) Drawing on our case-study files, we show how software should—and can—be made to play a vital active role in protecting an enterprise before, during, and after security incidents. Software can and should take active measures to safeguard customer data, business processes, and other sensitive data within the scope of the application. This approach, so far as we know, has not been addressed to this degree in other publications.

By taking a wide-angle view of security, we show how even parts of the company that are not on the firing line can nevertheless help make the company safer. (See Chapter 9, "The View from the Center," for example, for a discussion of the role of Human Resources in this effort.) And along the way we reintroduce, for the benefit of non-engineers (certainly a majority among today's developers and security practitioners) certain well-understood engineering principles, such as feedback loops and fidelity tests, into the battle. It's a fertile field—a frontier, really: an area of technical ground left unplowed because as a community of technologists we have not sufficiently considered the two disciplines of development and IT security together.

Moving Targets Are Harder to Hit

Much has changed while we wrote this book. Cloud computing has entered the scene in a massive way, as has mobile. When we started the book, Ken was carrying around one of those dreadful little "smart" phones with a physical keyboard. And then, along came a little company from Cupertino and one from Palo Alto that redefined mobile computing and forever changed the status quo.

These changes have had a huge impact not only on the computing world, but also on our project. A moving target is always harder to hit. But, in another and very real sense, they've underscored the need for such a book. We see small teams of "agile" developers these days diving into projects with sometimes reckless abandon, in search of the next great app. Although this innovation should be embraced and encouraged, we're also cautious when it comes at the expense of making some dreadful security mistakes.

One of the subgoals of this book is to help a new generation of software developers and IT security professionals avoid some of the horrible mistakes we've personally witnessed over the years.

Origins, Authors, Credentials

This book began over six years ago, in early 2008, with an observation by Ken van Wyk. A principal in a successful security consulting and training company, Ken had noticed an unaddressed need while conducting his "Secure Coding" classes for application developers around the world. Although his students were generally highly skilled at programming, and quickly acquired the technical defensive coding and diagnostic techniques he taught, the nature of "the threat" (and the tricks and mindsets of attackers) involved mostly new concepts. This lacuna hampered the ability of some students to anticipate what an attacker would do. At the same time, in conversations with enterprise security practitioners at some of the same companies he was teaching at, he found that some of the canniest firewall gurus lacked a basic grounding in programming beyond, say, some familiarity with scripting languages, but not higher level programming languages. These same gurus were often frustrated with developers who "didn't get it" when the topic was enterprise security, while they themselves often lacked familiarity with the "business logic" that was a prime mover on the development side of the house. As an old hand in the network security game, Ken saw that a yawning communications gap had opened in the field over the decades he had been working in it.

Ken approached Mark Graff, with whom he had previously collaborated on the successful tome *Secure Coding* (O'Reilly, 2003), and proposed writing a new book that would try to bring the two diverging fields back together. The working title was *Confluence*. Together Mark and Ken worked up a draft outline and successfully pitched the book to their first-choice publisher, Addison-Wesley.

We'll fast-forward the story here, to the point in 2011 when the two original authors realized they needed help if the project was ever going to make the bookstores (or web sites). After a round of recruiting (employing SC-L, the "Secure Coding" website inspired by the book and managed by Ken) and a diligent interview process, two new authors joined the project. Dan Peters (from a large security vendor) and Dr. Diana Burley (from George Washington University) filled out the team, supplying a critical mix of technical and field expertise, strategic thinking, academic precision, and a critical mass of time and energy.

Your four authors combine for about a century of experience in programming, engineering, education, communications, entrepreneurship, management, and security architecture. Our collaborative product examines *each* of these practices in the light of "confluence," the flowing together of heretofore divergent disciplines. Let's take a look.

Contents

We have arranged topics in this book in a logical order that roughly follows the high-level stages of a classic programming project, and chapter list below reflects that ordering. So after we take a detailed look in Chapter 1, "Introduction to the Problem," at the problem we are trying to solve—the divergence of technical streams that ought to be collaborating—we move into our own little development life cycle with Chapter 2, "Project Inception." While examining confluent design, implementation, testing, deployment, and maintenance, your authors illustrate various points in terms of real-world experience and provide a running example. Wrapping up, we recapitulate the material slightly from a new, integrated point of view, showing how a Chief Information Security Officer at a hypothetical company might undertake to put all the disparate pieces into motion to produce a confluent enterprise.

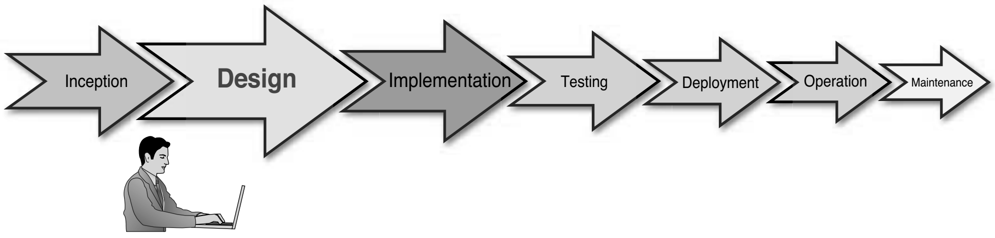
Generally, of course, we recommend that you read the chapters in order. This is a narrative analysis, not a technical compendium. We provide technical snippets, but mainly what we offer is argument and advice. And although we would be delighted to find that readers find a particular excerpt especially pertinent to their requirements, we have tried our best to write a book that can profitably be *read* in its entirety. It is our testament, in a way: an attempt to shine a light along a path we think leads to a safer and more resilient online world.

Summing Up

All these years have been a long time to write a book. We stuck with it because we think that there has been a bit of a wrong turn in the evolution of information security. We find ourselves in a cul-de-sac, but we think we see a way out. We hope you will agree. Let's get to it!

This page intentionally left blank

3 Design Activities



Let's get down to business by diving into some specific things we can accomplish together in our software development efforts. Design is a great starting point. Even for those of you following various agile (or other nonwaterfall) development methodologies, there's always some thought (if not documentation) given to the design aspects of a software project. As such, we're going to take a look at "design" in a general sense and include some aspects that you might or might not consider to be design work per se. These include requirements and specifications. And again, even agile practitioners should find value in these discussions.

But let's start with laying some foundations of what can and should be achieved—from a security standpoint of course—while we're designing our project. We know that a perfectly coded but poorly designed application can end up having egregious security defects. Perhaps more to the point, having an exceptionally clear picture of the application before implementing it, at least fully, can only serve to help. We realize that this concept smacks in the face of some development practices, most notably the family of agile development techniques. At the same time, we also support the notion of prototyping portions of code in order to better develop and understand the design itself. Such prototyping can take many forms, from rudimentary software, to wireframes, to notecard interactions with co-developers.

We discuss here two different categories of things to consider: positive practices to follow, and reviewing an existing design for security defects. Both of these are important to consider, but they're also very different in how we'll approach them.

Security Tiers

Before we proceed, though, we want to introduce a concept here; we refer to it as *security tiers*. We think it's useful to consider at least three tiers of security *readiness* as defined shortly. Note that we're in no way trying to define a maturity model here; it's simply worthwhile to consider a few security tiers, which will help steer us in the right direction as we proceed. Also, some projects might deem a low tier of security to be quite adequate, even when developed by teams that are highly mature in their software security practices. Thus, these security tiers refer to the state of the end product, not the maturity of the development team per se.

We're also not referring here to identity realms like one might find with single sign-on and other identity management solutions. In those situations, one has to pay close attention to transitive trust models in which an intruder can gain access to a user's session in a low state and use those shared credentials to breach a higher security state.

No, our concept here of security tiers is simply one of readiness within a single system. We believe that the concept is useful particularly at a design level to decide what security solutions to include and how to include them within an application system, whether it be simple or highly complex.

With that in mind, we'll keep the tier definitions to a simple *low, medium,* and *high* here and define them as shown in Table 3.1.

Table 3.1 Tier Definitions

Tier	**Definition**
Low	We think of low level here as meeting a bare minimum set of security standards for secure software. Basically, software written to this level ought to be able to withstand attacks such as those discussed in Chapter 1, but not necessarily contain any more security functionality per se. This is, of course, in addition to meeting its normal functional requirements.
Medium	At a medium tier, software not only should be able to withstand attacks, but should also be reporting and alerting security personnel appropriately about the nature of the attacks. (Of course, care must be taken to ensure that the event logs can never be used as a means of attack, such as XSS.)
High	At this level, our software can withstand attacks, report problems to security personnel, and be able to programmatically take evasive maneuvers against its attackers. The evasive maneuvers might include simple account locking (with due care to prevent intentional denial of service), user data encryption, recording of intruder information to be used as evidence, and myriad other activities. We think of this tier as a highly desirable state, particularly for enterprise software conducting substantial and valuable business.

These tiers will serve as a simple but fundamental basis for discussing different things that the development team and the security team can concentrate on during a project's design. Naturally, we'll see them again in subsequent chapters.

It's also worthwhile emphasizing that the low tier is at or above much of today's software in and of itself, because so much of what's running today is unable to withstand even relatively basic attacks.

We should also briefly talk about the rationale for having tiers in the first place. To illustrate our reasoning, let's use a common attack like cross-site scripting (commonly called XSS). For the sake of this discussion, let's assume that our application contains a customer registration form page that prompts the user for his name, street address, email address, and so on. Now, along comes an attacker who attempts to enter some maliciously constructed XSS data into one or more of the fields of our registration form.

If our software has been written to the low tier described previously, it would prevent the XSS data from causing any damage. The `<script>`

information will be stopped and the user typically be asked to reenter the malformed data. Perhaps this would even happen in the client browser by way of some JavaScript input validation.

However, in an enterprise computing environment, we might want our software to do something more. After all, a street address containing `<script>alert(document.cookie)</script>` (or some far more dangerous scripting nastiness) can only be an attempt to attack our software and not a legitimate street address. Particularly if our application's context is a business processing system, merely stopping an attack is just not adequate.

For a business system, we'd no doubt want to provide some information logging for our security team to look at, perhaps by means of an existing enterprise intrusion detection and monitoring infrastructure. That's where the medium tier comes in. Here, we'd make use of the security monitoring capabilities to provide useful, actionable business data to the security team. We'll discuss what sorts of things should be logged later in this chapter, as well as in Chapter 6, "Deployment and Integration," but for now, suffice it to say that we'd want the security team to have the data they'd need in order to take some appropriate administrative action against the application user.

And in some contexts, we might still want to take this concept further. When we detect a clear attack like the one in this scenario, we might want to have our software itself take some evasive actions. These might include locking the offending user's account, scrubbing the user's account of any privacy information, and so forth.

This scenario helps put in context how and why you might consider designing and writing a particular piece of software for an appropriate security tier. And, more to the point here, it's vital to start thinking about how you'll design these things into your software as early in the process as possible.

A great starting point when you're getting started down this path is to consult with your local IT security team and/or your incident response team. Since they are ultimately the "consumers" of the security components of an application, they absolutely need to be included in this process. For example, the contents of the logging information (tier 2) should be deliberately generated to support the incident response process. The type of information needed by a CSIRT (computer security incident response team) tends to be significantly different than traditional debugging logs, because it must include business-relevant data to find and catch an intruder. The principal purpose of debugging logs, on the other hand, is for developers to find and remove software bugs from a system.

It turns out that many of the decisions we make at this early design stage of a project, irrespective of any software development life cycle (SDLC) methodology we're following, have long-reaching ramifications from a security standpoint. For example, it might seem like a good idea to design and build some input validation all the way out at the application client code—perhaps for simplicity or to unburden the server with these seemingly trivial operations. Even though we know that client-side validation can be trivially bypassed, there are significant usability factors involved that might persuade us to do some of the input validation there—and then validate the data again on the server. For that matter, the server must never presume the client to be free of tampering. Quite the contrary, the design team and hence, the server itself must always assume the client can and will be tampered with. The client side code, after all, resides and executes entirely outside of the server's realm of control.

Software Development Life Cycle Methodologies

Software Development Life Cycle, or SDLC, methodologies prescribe the way software systems progress from conception to operation stage. There are multiple variations of these methodologies, ranging from very strict and formal waterfall-based models to extremely flexible and loosely organized agile variations. A new acronym, SSDLC (sometimes called sSDLC or SSDLC), has been created to refer to *Secure* SDLC, striving to explain how to transform normal product development life cycle to produce more secure outcome. (We should point out, in passing, that Microsoft calls its SSDLC process the Secure Development Lifecycle (SDL).)

In some business contexts, however, this approach might not be what we want and need. In particular, if a user does attempt to attack our client code, we've by design eliminated our ability to detect the attack and respond appropriately.

On the other hand, if we design the security mechanisms into the core of our software, we stand a significantly better chance of not only detecting an attempted attack, but being able to properly log what has taken place and to potentially take evasive action. We can choose, for example, to lock a user's account if the user has attempted to break our software. (In fact, this response might well even be mandated by the IT security or compliance team.) If that response is programmed in, procedures must be in place to review the triggering actions along with authenticating a user requesting that an account be unlocked. Consider the case when an attacker purposely triggers the locking of the enterprise's CFO's account. The CFO is going to want to get back into the system, but how do you verify that the requester is the CFO when he is yelling at you, and what if the actions were the CFO's and the logs indicate some insider financial manipulation?

All of these things are possible and feasible, but our design decisions can have a tremendous impact on how or whether we go about doing them. For this reason, we need to carefully consider our security design and make consistent architectural decisions that will properly support our business needs later.

On Confluence

In larger organizations one would expect to find more specialization, which is also true for security folks. Today, in our experience, one often finds IT security practitioners in large shops who elected that specialty early in their careers, and have little or no background in application design and coding. They are likely to be well versed in the latest attacks, and can be experts in setting parameters for firewalls and similar devices, and interpreting the log files. But several fine network security engineers of our acquaintance have no programming skills at all, whereas many others feel they are doing well to cobble together a functional PERL script to manipulate log files. This is the state of the practice today, and one of the drivers behind the need for confluence.

As we've said previously, one of the many key elements of success in addressing security issues properly from the beginning is confluence among the development and security teams. To be sure, many other stakeholders need to be included in the process as well, but none so clearly and comprehensively as these two. The overall process of teams and stakeholders' selection and considerations for doing it are explained in Chapter 2, "Project Inception." The thinking and consideration that should go into designing a business application are clear examples and opportunities of this confluence.

Microsoft's SDL process stresses this concept in a different way. Their cornerstones include both a Software Security Group (SSG) and people in a *security advisor* (SA) role, who act as the primary point of contact for all things security in product development. Note that, organizationally, SAs can belong either to the SSG or to the development organization, but work very closely with the security professionals. As such, they are advocates for the developers, not part of review or audit process that might at times be viewed as an adversarial role. We wholeheartedly support this concept and further believe it to be a vital success factor for developing a secure application system. Existence of such a formal security structure also helps with obtaining a senior management's mandate for following SDL practices and enforcing it consistently at all stages of product life cycle, because grassroots efforts often do not work well with security.

Key Responsibilities of the SA Role

The key responsibilities of the SA role include the following:

- Acting as a point of contact between the development team and the security team
- Holding SDL-related meetings for the development team
- Holding threat-model and design reviews with the development team
- Educating the development team on threats and best practices
- Analyzing and triaging security-related and privacy-related bugs
- Acting as a security sounding board for the development team
- Advising the development team on enterprise code guidelines, libraries, APIs, etc.
- Preparing the development team for security reviews and assessments
- Interacting with the information security team regarding CSIRT, monitoring, and other processes

We make it a point to describe key collaborative opportunities throughout this book, but perhaps the most important aspect of this exists here during the design process. No matter how rigorously your team engages in design activities, you're more than likely to be successful if you're able to reach out to all the stakeholders and include them as active participants in the design process. Having said that, care should be taken to clearly set roles and responsibilities, in order to prevent a "too many cooks in the kitchen" sort of situation. It is one thing to reach out to a stakeholder and solicit input; that is completely different than handing the helm over to the stakeholder.

In our experience, we've often found design review processes in which various stakeholders are included, but more often than not, the gating functions have been primarily business related and not security related. For example, can the application be developed within budget? Will it be delivered on time? Is the expense justifiable? Although this is all good and essential to do, we feel that security concerns need to also be included as early as possible, and that means including the security stakeholders in the process.

Requirements

Although many software developers these days eschew the practice of formally gathering and documenting their software requirements, there are many things worth considering at this earliest stage of development. Even if this is done only at an informal or "whiteboard" level, it can significantly help the team in understanding and capturing a project's security needs in addition to its functional needs.

We'll describe these considerations and steps here in several areas: abuse cases, regulatory requirements, and security requirements. Later, we will consider these requirements together with the security tiers we described earlier in the chapter. All these will come together as we discuss the topic of secure designs later in this chapter. Although the overall process is described as a team exercise, the role of the SA is extremely important throughout these activities, because he or she serves as both an anchor and a guiding force for all the participants.

Abuse Case Analyses

To start with, although abuse case analyses had been used in various ways for some time, McGraw's *Software Security: Building Security In*[1] provides us with one useful description of abuse case analysis. In essence, abuse case analysis looks at the intended functionality of a piece of software and seeks ways in which the software can be misused for evil purposes. As such, it is a review-based process to help us ensure that we're not building something that can be used to cause harm. That said, abuse case analysis can be a powerful means of finding problems with a project before it ever begins.

If the software will likely be misused or abused in a way the owner really does not want to happen, it is a serious problem.

Let's illustrate this with an example. Suppose you're the engineering team leader of your company's customer-facing web presence. One day, the vice president of marketing walks into your office and asks you to add a new feature to the web application: a mechanism for customers to subscribe to a new monthly newsletter the marketing department is launching. Simple enough; you can add a basic web form that asks the customers for their email address and perhaps some other contact data. After you have the information, you simply add the incoming addresses into a database of customers who receive the newsletter. All done? What could go wrong with this scenario? After all, all the functionality that the VP asked for is now complete, right?

Although it's true that this scenario fulfills all the functional requirements, there's a big problem. You probably recognized immediately that anyone could enter a "customer's" information and have her added to the subscription list. That's an abuse case. Heck, someone who really wanted to disrupt us could write a short script that would submit thousands or millions of addresses into our database if we're not careful. That's another abuse case, and one with obvious and really bad consequences. Now, let's take that further, to its logical conclusion.

If we recognize the potential for abuse, we'd want to prevent that from happening, naturally. A first step could be to add a security requirement to the functional requirement that might say something like "only verified email addresses may be added to the subscriber list." It's a good, actionable requirement. Our development team might implement that by sending an email confirmation to each address submitted for inclusion in the subscriber list. Now are we done?

Not so fast. Let's think a bit fiendishly here. If an email confirmation goes to the (intended) subscribers and requires them to verify that they want to be on the list, what could go wrong? Well, there's still an abuse case potential here. The mere act of sending out those confirmation emails could be disruptive. If an attacker bombards our subscription mechanism with fake but carefully chosen email addresses—say, at one of our key business partners—what would happen if our system then sends thousands and thousands of confirmation emails?

So it's not enough to send out a confirmation email; we have to ensure that our application is talking to a human, and not a script. There's another security requirement to consider. We note that CAPTCHAs are routinely used to address this issue. (CAPTCHAs are automated tests used to verify that a user is in fact a human. They usually show a distorted image of a word or phrase that an artificial intelligence would be unable to recognize but the user can read and enter correctly.) Nonetheless, let's add a security requirement such as "subscription requests may be issued only by human users of the system." See where this is going?

It's always best to consider abuses such as the ones we've described here before a system is rolled out into a production environment. But that requires the development team to be able to really think fiendishly, ignoring the mere functional requirements, and to consider how the system can be abused. It has been our experience that this can be a difficult leap for many developers. Security professionals, on the other hand, have been worrying about abuses like this for decades, and thinking fiendishly comes naturally to them. Invite them to participate.

In considering abuse cases, the following are some questions and important areas of concern to consider for each application. These questions are similar to those we'll address while doing a threat model, but let's consider them separately here while we ponder abuse cases.

➤ **How?—Means and Capabilities**

  • Automated versus manual

  In our mailing list scenario given previously, we saw an automated attack against a simple function. Often, designers consider a single use case with blinders on when thinking about how an application might be used. In doing this, they fail to see how the (usually simple) act of automating the functionality can be used to wreak significant havoc on a system, either by simply overwhelming it or by inserting a mountain of garbage data into the application's front end. Never underestimate the determination of a `while true do {}` block.

➤ **Why?—Motivations and Goals**

  • Insider trading

  Automating a user interface into an application is in no way the end of the myriad of ways an attacker can abuse an application.

Consider the human aspects of what an application will be capable of doing, and what sorts of bad things a maliciously minded person might be able to make of those capabilities. Insider trading should be a significant concern, particularly in publicly traded companies. Automation is, after all, a double-edged sword of sorts. We not only are automating a business function, but also might well be inadvertently automating a means for someone to attack a business function.

- **Personal gain**

Similarly, look for avenues of personal gain in an application. Ask whether a user of the application could use the information to "play" the stock market, for example, in a publicly traded company. This can be a significant concern in major business applications in enterprise environments.

- **Information harvesting**

Here, we look for opportunities for an authorized application user to gather—perhaps very slowly over time—information from an application and use that information for bad purposes. A prime example could include a customer database that contains information on celebrity or otherwise VIP customers, such as a patient database in a hospital where the VIP has been treated. That information could be very valuable on the black market or if sold to the media.

- **Espionage**

Although several of these issues overlap significantly, it's useful to consider them separately. Espionage, whether corporate or otherwise, could well be simply a case of information harvesting, but it's still worthy of separate consideration. Consider not just information like the celebrity database, but also company proprietary information and how it could be collected and sold/given to a competitor. What opportunities does the application being analyzed offer up to a user who might be persuaded to do such a thing?

- **Sabotage**

  Even in the best of economic climates, you'll occasionally find disgruntled employees who are bent on damaging a company for all manner of reasons. Their actions might be clear and unambiguously malicious—such as deleting files or destroying records in a company database—but they might also be more subtle and difficult to detect. Consider how a malicious-minded application user might be able to harm the company by sabotaging components in an application.

- **Theft**

  This one is sort of a catchall for things that weren't brought up in the previous ones, but it's worthwhile considering general theft at this point. Credit card account information is a prime candidate here.

Now, it's quite likely a software developer will look at a list like this and throw her arms up in the air in frustration, thinking it's not feasible to brainstorm something like this comprehensively. After all, it is fundamentally an example of negative validation, which we generally seek to avoid at all costs. Although that's true, there's still significant merit in doing abuse case analysis. Of course, the secret to getting it right is to do it collaboratively with some folks who are practiced at this sort of thing—like, say, the information security team.

It is also a good idea to consider separately the likelihood of an attack and the impact of a successful attack. These two things are quite different and bear separate analysis. Impacts can be imagined or brainstormed quite effectively, whereas likelihood can be more deeply analyzed, or even quantified.

Here's how the collaborative approach can work for analyzing abuse cases. After you've gathered a basic understanding of the functional goals of your project, invite a few key folks to take a look at the project and "throw stones" at it. You will want to ensure that all the interested parties are at the meeting; these should include at a minimum the business process owner, the design team, the information security team and/or incident response team, and the regulatory compliance monitoring team.

Tips on Conducting a Successful Abuse Case Study

To set the stage for considering abuse cases, consider holding a meeting among the key stakeholders: business owner (representative), IT security and/or CSIRT, security architecture, and developers. Before the meeting, be sure to distribute some basic design information about the application being considered. (But still bring copies of these documents to the meeting, just in case anyone “forgot” his copy.) This can be fairly high level at this point, but it must include a list of the business requirements and a graphic visualization of how the application should function, along with some basic narrative descriptions of the application’s components and what they do. Data flow diagrams can be useful here as well.

At the meeting, carefully introduce the purpose of the meeting to all the participants. Emphasize that you are only looking for misuse or abuse cases at this time. Your participants will invariably head down the SQLi or XSS path, but steer them back and focus exclusively on how the application’s core functionality could be abused.

To catalyze constructive discussion, orally describe the application’s functions through its phases: startup, production (including transactions, queries, or whatever else the application does), and how it shuts down. Next describe the normal use cases of each class of application user, emphasizing the types of information each user can access and what functionality she is presented with in the application.

You should be seeking areas of debate at this point. Seek questions like “What’s to prevent a user from copying all the customer records off to removable media?” When these questions come up, explore them, and be sure to keep the discussions civil and focused.

At this point, the best way to proceed is to describe the project to the assembled group. Discuss how the system will function and what services it will provide. You should be sure to list any existing security requirements that are already understood. At this point, run through a brainstorming session to collect any and all concerns that come up. The most important thing is to discuss the issues and enable—encourage even—the team to be as harsh as possible.

Take each security concern the group raises to its logical conclusion, and be sure to understand each one in detail. Make a list, for example, of any preconditions that would need to exist for an attack to be successful. So if an attack would need direct access to a server console, make sure that’s clearly annotated in the list of issues the group comes up with.

Next, take the list of issues and rigorously consider what security requirements could be added or enhanced to prevent the underlying cause from being exploitable. If an issue is not avoidable, consider security requirements that would enhance the ability to detect an attack if it does take place. A security requirement such as "all access to the application will be logged, with all user actions being recorded and monitored" can be useful, for example, in such situations.

It's also helpful to watch out for some common pitfalls with abuse case analysis. First and foremost, this process must be finite and has a clearly defined stopping point—which should be clearly communicated from the beginning to all participants. Any time you put a bunch of technical-minded folks together in a room, you're never guaranteed the outcome you expect. Engineers have a near-overwhelming inclination to digress in ways you can't begin to fathom. Abuse case analysis is no exception to this. Expect them to discuss low-level technical details such as buffer overflows, cross-site scripting, and a myriad of other things that just aren't relevant at this stage.

To get value out of abuse case analysis, it is absolutely vital to facilitate and guide the brainstorming process carefully but firmly.

Asset Inventory

We've also found it useful to start at this point to generate an inventory of the sensitive assets the system will need access to. This inventory should include such things as customer records, passwords, and encryption keys, as well as the high-value functions in the application. If applicable, consider prioritizing the inventory in terms of value to the company. Although a large enterprise might have to set up a large corporate project to identify key assets (a customer database, for example), security-conscious folks in smaller places will have their arms around those assets all the time.

In the Microsoft SDL approach,[2] they describe a process called threat modeling. An asset inventory is absolutely vital to doing threat modeling, but it's not the same thing. We're trying to articulate here a very clear understanding of everything of value in our application. In other words, what are the targets an attacker is most likely to go after? If we can build a solid understanding of what the targets are and prioritize them in a relative manner (say, low, medium, and high business value, recognizing that one company's "low" could well be another company's "high" and so forth), then we can also understand what can and should be protected, and how much effort we should put into protecting each.

As we said previously, an application's assets can include important data, but also functions. For example, many applications have an identification and authentication mechanism. Since these are by nature accessible to unknown attackers, they almost always should be included as high-value targets in an asset inventory. That will help us later in allocating the necessary resources for reviewing and testing those modules.

Now, although developing an asset inventory isn't something that can or must be done during a requirements process per se, it's still a good idea to start thinking (and documenting) this as early as possible. Microsoft's SDL process starts this step early as well.

Regulatory Requirements

Next, we should consider the security-related regulatory requirements that our software is going to have to operate under. In many industries today, our business systems are required to conform to myriad security laws and guidelines. This is particularly true in publicly traded companies as well as certain highly regulated industry sectors such as financial and insurance services, pharmaceutical and healthcare, and public utility companies.

Additionally, companies that operate internationally might have country-specific regulatory and privacy requirements to comply with. Naturally, this can greatly complicate the security requirements process. In some cases, the application itself might need to operate differently based on where the customer, employee, or other user is located.

And especially in these extremely complex environments, it is commonplace these days to find corporate-level compliance officers or at least a compliance monitoring team. Often, the compliance team will organizationally fall under the CIO, COO, Audit, or even General Counsel's office.

Step number one in this part of the design process is to seek out the person or department in charge of compliance monitoring and engage him or her in the process. As a starting point, specifically look for issues such as the following:

- **Data or information that needs to be protected for privacy**

  Many business systems are required to safeguard the privacy of customer data, Social Security numbers, credit card numbers, and so on. These are vital to the security of the application, and the sooner the development team is explicitly aware of the requirements, the better off everyone will be. Find out the specific privacy issues for each data element. In some circumstances, it might also be useful to consider privacy requirements for various markets, even if a product isn't (yet) marketed in some of those markets. Considering those requirements now might well save us substantial grief later, should the company decide to expand into those markets.

- **Data retention requirements**

  Several U.S. Government bureaucracies—and no doubt many others—have stringent requirements on data retention, covering things such things as email and transaction records. It is important to gather all of these requirements and investigate how the application itself can help support them, instead of simply dismissing them to the data center staff to implement. As an example, consider the data retention requirement the Securities and Exchange Commission in the U.S. imposes on broker-dealers. It's called "Rule 17a-4," and it dictates that certain records (trade blotters, order tickets, trade confirmations, and much more) be preserved in nonrewritable and non-erasable format for specified periods. For "communications that relate to the broker-dealer's business as such," the retention requirement is three years.[3] If your app will operate in a regulated environment, we recommend you get expert help to ensure that you facilitate appropriate data retention.

- **Data or processes that require special reporting**

  Many security regulations have explicit requirements for reporting particular types of data access and such. Credit card transactions, for example, might be required to be logged (but not with customer-sensitive information in the logs) under the Payment Card Industry Data Security Standards (PCI-DSS) requirements. There might well also be breach reporting requirements for many applications and the jurisdictions in which they operate.

- **Entity identification or authentication requirements**

  Some sensitive application environments are required to meet minimum standards for strong user and/or entity identification and authentication. PCI-DSS again provides us with ample examples, such as in Requirement 8.3, which says, “Incorporate two-factor authentication for remote access (network-level access originating from outside the network) to the network by employees, administrators, and third parties.” Portugal’s Digital Signature of Invoices law represents another example of Entity Identification requirement; it attempts, among other things, to bind invoice documents to the software that was used to create it.

- **Access control requirements**

  Sensitive data or functions within an application can require additional access controls for read and/or write access. These are often designated in industry requirements such as PCI-DSS once again. PCI-DSS Requirement 7 states, “Restrict access to cardholder data by business need to know.”

- **Encryption requirements**

  In addition to access control, many sensitive data elements need additional privacy and integrity protection using cryptographic controls. PCI-DSS 8.4, for example, tells us, “Render all passwords unreadable during transmission and storage on all system components using strong cryptography.” Note here two things: that the requirement covers passwords while both at rest and in transit, and that it leaves open significant options in how to implement the standard, even though it does define “strong cryptography” in the document. It is nonetheless actionable and exactly the sort of security requirement we should be looking for. Further, it is the sort of requirement that can and should evolve with time, as cryptographic algorithms are retired, new practices discovered, and so on.

- **Change-management requirements**

  Many highly regulated industries, such as the pharmaceutical and healthcare sector in the U.S., have rigorous requirements for change management of production business data processing systems. Even though change management is not something that a software developer always has a direct role in, it is still important to be aware of these requirements and to adapt the software practices to fit into them. One exception here regarding change management has to do with source code repositories. Strong access control for both “read only" and "read write" permissions in a source repository should be emphasized, even if only to safeguard things like comments in source code containing sensitive information about a project. The same holds true for a project's bug tracking system.

It would be easy to assume that some of the topics in the preceding list are "someone else's job" and thus outside of the scope of the development team's efforts, but that would be unfortunate. Although some of these topics are in fact someone else's responsibility, in order to be effective, there must be a clear interface between them and the application itself. The more cohesive the bond between these requirements and the development team's efforts, the better the end product will be. Put another way, it should be clear by now that there are many stakeholders in the overall security of a typical business application, and they should all be consulted and included in the planning and implementation.

Security Requirements

In the preceding section, we discussed regulatory requirements. These tend to be driven by governments, industries, or other standards bodies—but nevertheless external to the company that develops or owns the software. Let's now focus on some internal requirement issues.

From the development team's perspective, the thought process to go through is largely similar: Seek out the appropriate stakeholders and engage them in the process to find out what their security requirements are. The primary difference is the stakeholders themselves. Whereas we looked to the compliance team previously, now we should be looking at the internal information security group directly in this part of the process.

In looking internally for security requirements, there is another place to take a look at as well: internal security standards. Much like their government and industry counterparts, internal standards will often include requirements for such common security mechanisms as authentication, passwords, and encryption. Although they are typically more detail-oriented, quite often they fall short of being truly actionable, but they're still a good place to start looking.

The sorts of things to look for at this point include the following:

- **Identification and authentication requirements**

  Many large corporations have application security policies and guidelines that include identification and authentication. Some go so far as to designate tiers of sensitivity for applications—generally three to five levels of security, such as "level 1: customer data," "level 2: company proprietary information," and "level 3: public information." Note: These levels are for illustrative purposes. It is often the case, for example, that customer data covers multiple data types and corresponding sensitivities. Password and authentication factor guidelines are also commonly found. They'll typically include minimum length, frequency of password changes, and character set requirements for all passwords and such, along with any multifactor authentication requirements for the most sensitive applications. It is not uncommon for a company's internal security standards or policies to prescribe specific guidelines for proper credentials storage, specifying certain hashing and/or encryption algorithms and when those should be applicable. The development team must follow these guidelines if they exist. If they don't, now is a good time to define them for this project and those in the future.

- **Event-logging requirements**

  Despite the fact that many enterprise data centers have existing architectures in place for centralized event logging, it has been our experience that most event logging actually takes place at the operating system and web/app-server level. That is, we've rarely seen application-level logging that is truly adequate for the incident responders to properly do their jobs. We'll discuss this topic in much more detail later, as part of Chapter 6, but for now, let's at least ensure that the development team is fully aware of any and all event-logging requirements and infrastructures that are in place. We say this in the plural because many enterprise environments log both operational and security event data, and they often separate those two types of logs quite substantially.

- **Disaster recovery and business continuity requirements**

  Most large enterprises do significant disaster recovery and business continuity ("DR/BC") planning these days. Although much of this has to do with natural disasters such as hurricanes, floods, fires, and earthquakes, it is still important to engage in conversation with the folks doing this planning. In particular, look for requirements around alternative data centers (or hosting services, and so on) and other contingency planning in order to understand how your application will be able to support that type of requirement. The plans often include requirements for rotating to alternative "hot" or "warm" data centers with specific minimum downtime requirements and such. These things are often considered outside the direct scope of the application development process, but it is important to have at least a minimum understanding of the requirements. There might well be, for example, requirements to be able to have an application seamlessly, without downtime, of course, switch to different event logging or other infrastructure servers. These can have a significant impact on how the development team designs and implements many such configuration settings.

- **Incident response requirements**

  Increasingly, corporations have in place incident response teams, either in-house or outsourced. These teams are generally faced with one or more of the following challenges when an incident occurs: diagnose the problem, contain and/or stop the incident, investigate (or support the investigation of) an incident, or perform a damage assessment after an incident has taken place. In pretty much every case, the common denominator and the "lifeblood" of the incident response teams is having a clear and accurate situational awareness of what is going on or what did go on inside the affected business application. This invariably leads to event logging.

  Now, although we've already raised the event-logging requirements previously, incident response requirements can be quite different from ordinary event logging. For example, the incident response team often has a need to capture and store log data and maintain a chain of evidence so that the information will subsequently be useful in a court of law. They also often need to assemble from disparate information sources a clear picture and timeline of what an attacker did (or attempted to do) during an incident, which requires log data to be rather detailed across all the components and layers of a complex business application. This can be a daunting task under the best of circumstances. Timelines can be better reconstructed if components generating log entries have synchronized their times. "System time" policies are becoming even more important as enterprises utilize distributed systems and cloud computing environments spanning multiple time zones, and so on.

As such, it's quite possible that the incident response team will have quite a "wish list" of things they will need from your application when doing their jobs. That wish list is generally borne from experience and operational need when it comes to performing their jobs as rapidly as possible. In reality, their normal mode of operation is to adapt to and to work with the information they have available, but what better time to ensure that they'll have what they need than during the early phases of developing an application?

The best way to do this is to gather a clear understanding of the incident response team's use cases for how they will need to interact with your application during an incident. Meet with them as you would the business owner or user community to find out how they'll use your application.

- **Account management requirements**

  Another commonly set of functional and security requirements can found in account management practices. Corporations often have guidelines and policies for user and employee accounts, as well as for third parties, contractors, consultants, and so on. Employee accounts on enterprise applications might need to be synchronized with employee records in the Human Resources department, for example, to ensure that accounts for departing employees get deactivated when an employee leaves the company. As you might imagine, we've seen many mistakes made in this area over the years. All too many business applications are written in the absence of any means of verifying employment.

  You're likely to find several relevant stakeholders when it comes to account management practices. You might find, for example, that Human Resources can contribute substantially, in addition to the IT Security department for their policies on opening and closing application accounts. But don't stop there. Even the incident response team might be able to contribute with requirements for deactivating accounts during incidents while maintaining their data for forensic analysis or evidentiary purposes.

- **Access control requirements**

  In our experience, access control requirements for applications tend to be rather superficial even in larger corporations. It's not uncommon, for example, to find access control statements that designate user-class and administrator-class users of an application and what they should be allowed or not allowed to do. However, it's not common to find access control requirements that go beyond these simple one or two dimensions.

  That might well be quite adequate for many applications, but it still bears consideration at this early stage. Among other things, it can open up a significant set of possibilities to have more rigorously defined role-based solutions for some more complicated applications.

  With that in mind, the most relevant stakeholder for gathering access control requirements is most often the business process owner—the person who is responsible for the business functionality of the application itself. In talking with the business owner, it's important to listen for language that would lead you to need more stringent access controls than simple user/administrator accesses. Listen, for example, for language like "anyone in accounting should be able to do 'x'," whereas "those in HR should be allowed to only view the information, not change it."

  Irrespective of which departments and stakeholders are considered, a well-designed access control system and policies should be based on the venerable principle of least privilege.

- **Session management requirements**

  Session management is a big issue at a technology level for web-based applications, but it's still relevant for many other application architectures as well. The sorts of things to look for with regard to session management should include timeout periods for inactivity, time-of-day restrictions, location restrictions, failover capabilities, and so forth.

  As with access control requirements, the most likely relevant stakeholder for these issues tends to be the business owner. And the way to approach the topic of session management requirements is to seek use-case scenarios. (These might well already be defined, so be sure to read up on what information has already been gathered.)

At a more technical level, there also might be company standards or guidelines on how to implement session management in an application, particularly if the application is web-based. Enterprise data center environments often make use of either single sign-on or other centralized authentication services and APIs, which are equally important to be aware of and make use of when designing an application.

Hopefully, these standards include security guidelines on such issues as session fixation, safeguarding session cookies, and cookie contents. Great care, too, should be taken in cookie generation. Persistent cookies should be rigorously encrypted, for example. This type of technical session management requirement is not at all likely to come from the business owner, but rather the security team, because these are things that often are discovered during security reviews.

- **Encryption standards**

  Particularly in regulated industries, there are often policies for encrypting sensitive data. Sometimes, these requirements don't come from external regulations, but from the security department directly. At a bare minimum, it is important to find out what these standards are and to conform to them. Most often, the guidelines serve to specify what encryption algorithms are acceptable for particular types of application data. In most cases, they explicitly and strictly ban any attempts to come up with "homemade" cryptographic functions, requiring teams to rely on existing and vetted algorithms and implementations instead. What is often missing in encryption standards and requirements is detailed information on how the entire crypto system should work, such as key generation and management. Those details are typically up to the developer, and great care must be taken in how these things are done.

  These standards are all important, of course, but we should point out that there is still plenty of room to make mistakes. In our experience, far more encryption problems arise from poor key management practices than from selecting algorithms that aren't up to the task—and very few encryption standards even address the topic of how best to do key management.

Password storage is another topic that should be taken up in encryption requirements. It is recommended that password storage standards use a one-way hash function approach that combines the password with other information such as the user account identifier. In this manner the same password used by different accounts will not result in the same password validation value (hash). Being a one-way hash, it should also be computationally expensive to derive the password from the hash value. This helps to minimize the impact if the account and password values are stolen from their storage on a server component.

- **Change-management requirements**

  Most even moderately mature enterprises have documented processes and procedures for handling changes to production applications. For the software developers, the key is to know how best to interact with that process and work within its boundaries. As with disaster recovery and business continuity, these requirements can have an impact on how best to design and implement an application. For example, if an application must maintain login credentials to connect to a database server, it's generally best to keep those credentials in a properties file (of course, protected, as, hopefully, specified in credential management and encryption standards) and never hard-coded in the application's source code. Apart from the security vulnerability introduced by hard-coding login credentials in an application, keeping them in a properties file often makes things easier from a change-management standpoint. This is because changing a properties file on a production system is typically far easier than changing the source code and rebuilding an application. So, as a starting point, the stakeholder to look for on this is generally in either the CIO or the COO environment, or perhaps IT and IT security, depending on who sets the change-management processes in your organization.

- **Patching requirements**

  All software has to be patched periodically. Patching and updates follow very formal processes when external customers are involved, but for internal-only products these rules can be somewhat less rigid, although they need to take into account all of an application's components, from its operating systems through its libraries, frameworks, and others. In any case, the development team should take future patches and possible formal requirements around this process into consideration early in the design phase.

Where things become more complicated is with updating software components that require rebuilding the underlying application software. It is not uncommon for organizations in an enterprise to have an unclear or otherwise unrealistic understanding of which organization is responsible for deploying specific patches. For example, installing operating system patches is generally fairly easy, whereas replacing a software framework requires a complete rebuild of an application. In these cases, the patching is best not left to the IT operation staff.

Also, it is worth noting here in passing that large enterprises are increasingly insisting on security requirements, including patching, in their contract verbiage with software vendors.

As you can see, this list is far more internally focused than the one in the preceding section. It's no less important, however. It's also worth noting that it's more than likely you'll find that no standards exist for many or even all the items in the list just given. In that case, it would be far too easy to dismiss the topic and continue in a "business as usual" mode, but that too would be unfortunate. Instead, we suggest you consider it an opportunity for collaboration between the development team and the security team to put together a meaningful and actionable set of guidelines and requirements to address this list (and more).

Bringing It All Together

Many of you reading this might well feel overwhelmed at this point. We've just laid out a highly ambitious list of things to consider when gathering the security requirements for a business application. The list is daunting, we agree. However, the news isn't all bad. Let's consider some of the positive aspects of what we've been covering in this section.

For one thing, it's highly likely you won't need to do all of this with every application. There's an economy of scale to be found here. So if, like many organizations, your organization handles multiple business applications, then you can definitely expect to see a reduction in the level of effort with each passing application. Also be cautious of single sign-on environments where a weakness in a relatively low-risk application can result in stealing credentials to compromise a higher-risk application. Either way, the first project that embarks on this path must have executive support because they are more likely to see the return on investment later as other enterprise application projects arise.

Also, remember the notion of having a security advisor working with development teams? Well, here's an opportunity for the security advisor to shine and prove his value to the development effort. The security advisor should, among other things, be expert at all the security requirements an organization needs to conform to, external as well as internal.

At the very least, the organization's security team should be able to provide a significant list of applicable requirements, laws, and guidelines that will need to be followed on an ongoing basis. This list will need to be periodically updated since many of the standards change over time, and the list should include a resource library with searchable documents for each set of requirements.

So although doing this correctly would require a rather significant initial outlay of effort and possibly electronic resources, every application project should be able to benefit from that effort, reducing the per-project costs significantly over time. This, by the way, is a compelling argument for at least some level of centralization of software development infrastructure in an organization.

With that in mind, it has been our experience that the best way of collecting security requirements is via meetings and interviews with the stakeholders. Ideally, this will be done with the project's security advisor on hand, but even if your organization does not have a security advisor, it's still quite achievable. Here are some procedural considerations.

- **Do your homework**

  No matter how eager your stakeholders are to contribute and help, they'll always appreciate it when you spend some time in advance and do some preparation. Start by doing some online research and information gathering about the external and internal regulations you believe are applicable to your application. Download and read the latest version of all of them. If there are preproduction versions of any of these standards in the development pipeline, get those as well. It's quite likely those will become relevant to your application after they're released, so you'll want to be aware of what's coming along, even if they're not currently in their final form.

Make a list of the regulations, standards, laws, and so on, including the internal security policies and guidelines that are applicable to your project.

While reading through all the standards, make a list of questions.

Use this time also to ensure that you deeply understand the business intent of the application. Know what it is intended to do and what it is not intended to do. Ensure that the stakeholders agree to this as well, and understand the pitfalls of mission creep. And be realistic that some mission creep is simply inevitable.

Make a list of all the stakeholders you'll need to talk with. These might be individual people, or perhaps roles or departments (e.g., General Counsel or Compliance Officer).

- **Start with abuse cases and asset inventory**

  Using the most preliminary and basic set of functional requirements for your application, go through an abuse case analysis process for the application as described earlier in this chapter. Some of the issues uncovered in the analysis could well turn out to be addressed in the various security requirements, but it never hurts to spend the time to really understand how your application might be misused after it is deployed. Plus, it's been our experience that understanding the abuse cases helps build your own knowledge of the application and what aspects of it really need to be well protected later.

- **Invite the stakeholders**

  Depending on the nature of your questions and agenda, you might end up doing one-on-one interviews with the various stakeholders, or you might end up inviting them to one meeting. Whichever works best for you, invite all the relevant stakeholders to participate in this stage of the application development process, again being cautious to avoid the "too many cooks in the kitchen" problem.

- **Brainstorm and refine**

  With your stakeholder(s) gathered, ask your questions and dive deeply into the answers. Where answers seem vague, push to make them explicit. You want to seek clear and actionable answers here wherever possible. Toss out topics for discussion that perhaps the stakeholder hadn't considered. You might need to illustrate things via examples and case studies to make them clear. You're also likely to be questioned about the likelihood of something bad happening. Has it happened before? When? And so on.

Gathering the security requirements for an application can seem like a lot of effort indeed. However, getting this step done well will undoubtedly have significant payoffs throughout the development effort, irrespective of the software development methodology your organization follows. And again, we haven't even begun to discuss the security tiers we mentioned early in the chapter yet.

We've placed a lot of emphasis on requirements gathering because mistakes made now can have a multiplying impact later. Neglecting something like a policy on encrypting customer data can result in massive reengineering if we get "surprised" by the requirement after we've designed and built the application. Things get more dynamic and exciting when Agile processes (and variations) are introduced, because they tend to be less planned and more flexible. As the authors have frequently observed, Agile teams skip documentation updates altogether during these so-called springs (literally living by the motto "code is the best documentation"), which creates quite a lot of issues for security analysis. As a result, the security requirements have to be continuously reintroduced and readjusted based on the current project planning—most likely, as a mandatory integral part of weekly sprints.

There's a secondary benefit from going through a rigorous security requirements gathering process early, and it's one we haven't discussed yet. By engaging all the application stakeholders in dialogue long before starting to actually design or develop any code, you're including them in the process rather than asking them to just review and accept your work later. It's rare to find the individual who doesn't appreciate that approach.

Specifications

If you've done a good job at collecting and synthesizing the security requirements, turning them into specifications—if indeed your process even calls for this step—should be relatively smooth sailing. In essence, the requirements speak to the *what,* and the specifications speak to the *how.* We've seen many examples of the specifications being rolled into the design of an application itself, and many organizations don't bother with articulating a separate set of specifications above and beyond the requirements they've already gathered.

For our context here, we're not going to presume any specific development methodology, but we do want to make sure that our readers understand the benefits that can be gained from documenting the project's specifications.

So let's consider a few examples from our security requirements given previously.

We described a PCI-DSS requirement (8.3) that says, "Incorporate two-factor authentication for remote access (network-level access originating from outside the network) to the network by employees, administrators, and third parties." This is already fairly specific language, but we might fine-tune it for our own application by saying something like, "Use [hardware token] and associated authentication server for authenticating all nonlocal application logins." We also want to ensure that we have a degree of traceability from our requirements to our specifications. Numbering each, and being consistent with the numbers, can be enormously helpful.

Similarly, we cited another PCI-DSS requirement that says, "Render all passwords unreadable during transmission and storage on all system components using strong cryptography." Here, the specification verbiage, which should be driven by the enterprise's security standards, could become something such as, "User passwords must be transmitted in TLS 1.1 encrypted sockets using (minimum) 128-bit keys and cipher suite [example] and stored on the LDAP server in hashed format using SHA-384 (minimum)." This example, provided here merely for illustrative purposes, clearly shows how a relatively open requirement statement becomes specific to a local environment.

At a minimum, it is worth going through all the security requirements you've collected, and localizing them with specific details on local policies for such things as password lengths and encryption standards.

From a security perspective, we are obviously placing less emphasis on this aspect of the design process. The reason for that is that the core important details typically surface during the requirements phase and not so much in the specifications. That's not to diminish the value of documenting an application's specifications. Quite the contrary, details such as those we've listed here are important, but they also tend to be a fairly direct and simple mapping from the various requirements guidelines and policies.

There is one important exception to this, however: contradictions. Particularly in heavily regulated or complex environments, it's not uncommon to find contradictory guidance in various security requirement sources. One source might, for example, indicate using an AES-128 encryption algorithm, whereas another says Blowfish is adequate.

The rule of thumb to follow here is to go with the higher level of security. But that's not always immediately obvious in all cases, as with the AES versus Blowfish example. In a case like this, it's probably best to seek policy clarification from the IT security organization, or better yet, a professional cryptographer.

Design and Architecture

Now, with our security requirements and specifications document, it's finally time to start considering the architecture and design of our application—or for our purposes here, the security aspects of the design. If you've been following our advice on collecting requirements, you're likely to find that many of the security details in the design will essentially "write themselves." Okay, that's an exaggeration, but at least many of the architectural security decisions should be made easier from a good collection of security requirements.

These decisions will also be heavily driven by the expectations for security tiers introduced earlier in this chapter—in particular, we're faced with architectural decisions of where to place specific security mechanisms for each security tier. If "tier 1" is your target, the decisions are likely to be quite simple; you can accomplish most of tier 1 within an application's presentation layer, especially server-side, with a relatively small amount of back-end coding being required.

On the other hand, if your target is tier 2 or 3, you'll want to give careful consideration to the security mechanisms you employ, and where they should be placed within your application. Building (essentially) intrusion detection functionality into an application is generally best suited for the business logic layer of an application, but there will still be some security functionality in other areas of the application, from the presentation layer through the data layer.

The important thing is to consider the architectural ramifications carefully, and then implement consistently throughout the application. Mixing and matching architectural components for convenience or familiarity is not a recipe for success.

So it's on to considering the design or architecture of our application. We're big believers in prescriptive guidance of positive practices, rather than (just) reviewing a design for flaws. So we'll start there.

Prescriptive Design Practices

Perhaps the most important prescriptive practice to follow in designing secure applications is relying on common well-tested infrastructure and reusing already vetted design patterns, but that presents a "chicken and egg" sort of dilemma. By that we mean repeatedly using a set of design components that have proven themselves to be secure. Additionally, it's useful to use design checklists that verify certain positive compliance aspects of design components. Let's consider these things in some detail.

But first, let's briefly take a look at the origins of these practices. Our secure designs should be built on top of sound architectural principles, such as those described by Saltzer and Schroeder in the 1970s.

Saltzer and Schroeder's "The Protection of Information in Computer Systems"

This pivotal work by renowned engineers Jerome Saltzer and Michael Schroeder introduced several data protection principles that are as relevant today as they were when the paper was first written in 1975. These include the following:

- Economy of mechanism
- Fail-safe defaults
- Complete mediation
- Open design
- Separation of privilege
- Least privilege
- Least common mechanism
- Psychological acceptability

Knowingly or otherwise, we make use of all of these design principles in virtually every aspect of our software today. Every person involved in building secure software should be intimately familiar with these principles and how to apply them.
They should also of course incorporate the security requirements and specifications we've spent so much effort in collecting and documenting previously.

From there, we should look at various security aspects of our design for construction soundness and turn those into checklists that can then be reused in later projects or verified and validated in the current project. Ideal targets for such checklists should include the following set of focus points:

- **Identification and authentication**

  Things to look for in a strong "I & A" mechanism include con–forming to corporate standards for username and password—minimum length, acceptable character sets, and so on—but extend well beyond that. Software designers should also ensure that all sensitive information is adequately protected while at rest as well as while in transit. They should also ensure appropriate credentials management practices, that is, that all passwords are securely hashed and salted, and then stored into a repository. The credential repository itself should conform to any standardized architecture that is in place. Login credentials should also never be exposed in the URL field of a web browser, via a `GET` method; they should instead be embedded in the HTTP request body as a `POST` parameter. For that matter, it is entirely possible that your enterprise already has a standardized identification and authentication, or identity management, architecture, and you'll need to make use of that. That is all well and good, but the preceding criteria should still be considered carefully.

- **Key management**

  Although the topic of key management is substantial, private and symmetric keys, used by an application, require specific handling and have to be adequately protected according to the IT/IS guidelines. The most frequent solutions here are the use of key stores, file permissions, or specialized hardware tokens for highly protected systems.

- **Access control**

  This one is more problematic to find through a simple checklist process per se, but there are still some things that can be verified. The basic principle to doing access control inside an application is to ask the question of whether a user, an entity, or a process should have access to the data or function it is requesting. That is, all data and function calls should in fact be designed as requests that can be authoritatively answered.

As such, the thing to look for at a design level is a centralized access control mechanism that enforces policy. The policy itself will be set—perhaps dynamically—elsewhere, but internally, there needs to be a means of answering the question of whether a request should be permitted.

What makes this problematic is that every access needs to follow this requesting methodology. We'll discuss this in more detail in Chapter 5, "Testing Activities."

- **Boundaries**

  Every application of even moderate complexity has numerous boundary layers. They can be between components, servers, classes, modules, and so on, but at a design level we generally have a bird's eye view of all of them—at least if we're doing it right. From a security standpoint, boundary layers offer a positive opportunity to verify good practices like input validation and access control.

  Most risk assessment methodologies implicitly map out boundary layers in their threat modeling process by defining security zones (aka trust boundaries). Each zone is then studied for the risks it poses.

  Looking at an application's boundary layers is similar, but generally offers a slightly more data-centric perspective on things.

- **Network connections**

  Network connections, including both physical and VPN, are essentially a single boundary between components, but we list them separately here because they offer different types and levels of security controls. For one thing, in most enterprises, the networks themselves are operated by the IT organization. And further, network-level security controls tend to be outside the direct scope of the application itself, but nonetheless can be useful at independently enforcing some policies.

  For example, a network layer between an application server and a database can enforce permitting only SQL network traffic between the two components. Although the network can't often do much more than that, it does provide us with some useful controls that help us enforce some of Saltzer and Schroeder's design principles—namely, compartmentalization, graceful failure, and least privilege in this case.

- **Component interconnections**

  These are simply another form of boundary layer, but much like the network boundaries, separate application components can offer up different types of opportunities for security controls.

- **Event logging**

  Event logging is a big topic for application developers, and it is one that is almost always not well understood or adequately implemented. The key point that most developers don't get is that the customer for event logging should be the IT security or incident response team. As such, it's vital to consider their use cases with regard to event logging. More often than not, application event logging mostly contains debugging information. Although that information is useful for debugging purposes, it's not all that's typically needed when responding to security incidents. In addition, the security team often needs more business specific logging in order to determine the "who, what, where, when, and how" sorts of things they need to do.

  From a design checklist standpoint, you should be verifying that the security team has provided their log use cases and that those use cases will be incorporated into the application's design. For details and examples of such use cases, see the discussion in Chapter 6.

- **Session management**

  Most modern platforms and application servers these days provide more than adequate tools for building robust session management into applications running on them, but mistakes can still happen. We'll discuss this further in Chapter 4, "Implementation Activities," but for now, let's ensure that the available infrastructure for doing session management will be used for our application.

- **Protection of sensitive data**

  Enterprise applications carry all sorts of sensitive data these days, and it's up to the developers to ensure that the sensitive data is being properly protected. As with protecting any secret, it's vital to consider sensitive data at rest and in transit, because each state carries with it a different set of protection mechanisms that should be considered.

- **Data validation**

  Any time our application has to make use of an external service—command-line interface, LDAP directory, SQL database, XML query engine, and so on—we have to ensure that the data being sent to the service is safe, after we've mutually authenticated those components, of course. There are several key concepts in that sentence. First off, we can't assume that a service we're calling is going to adequately protect itself. Second, we have to ensure that the intent of our service request is immutable from change due to whatever data we're sending to the service.

  Essentially, we have to do proper input validation for the current module and output encoding of what will be sent to the service before making the call, and we have to do it in the context of understanding what the next service call is intended to do. That's a pretty tall order, because encoding for a database call will be different from the one for LDAP query, for instance.

Design Checklists

It's a good idea to keep a checklist or two readily available to the design team. These checklists should cover basic topics to ensure they are appropriately represented in various components of a system's design. It is also vital to be cognizant of the fact that checklists cannot capture everything, and can and should evolve over time and quite possibly per application. Still, they can serve as a good starting point for catalyzing thoughts. Sample checklists should include the following:

- Establishing connections

  This checklist should step through all the things one must do in establishing a connection between two application components, such as ensuring mutual authentication and ensuring transport layer protection of data.
- Authenticating users

  This list should point to standard design components for authentication, such as enterprise authentication services and how to invoke them.

- Managing sessions
- Securing data at rest
- Securing data in transit

…and so on. The point is to promote consistent conformance to established design patterns.

Oh, and you'll want to carry many of these checklists forward as coding guidelines (with rigorous annotations) as well. We'll discuss that in the next chapter.

As you might imagine, the earlier list is by no means a comprehensive one, but it does represent a pretty common list of application aspects. For each of these common problems, we should put together a checklist of issues to ensure that we have properly addressed them in our own designs. Plus, since this list of things is pretty common, we can use it as a basis for some design patterns that we'll be making repeated use of.

Now, some of the elements in the list aren't necessarily discrete application components—say, for example, protecting sensitive data—but they are all things we should address as we consider the security aspects of our design.

It's also worth giving careful thought to the feasibility of security aspects of a design. It is a common mistake to overengineer a design and basically attempt to protect everything all the time. The problems with this approach are numerous. For one thing, we're likely to end up with a solution that is too costly for the problem we're trying to solve. It might also be too complicated to be successful, or building it that way might simply take far more time than we have available. For that matter, it is also common to underengineer a design due to tight timelines, budgets, and so on. The key is to find the right balance, as in so many things.

So we have to make compromises, but we have to be able to do so in a principled, businesslike, and repeatable sort of way. That is, we have to have sound justification for the design decisions we make or don't make. That's where a risk management methodology comes in.

Risk management is one of McGraw's three pillars of software security,[4] and it helps us make decisions with confidence. Without a sound way of considering business risks, we're inevitably going to make the wrong decisions by accepting risks we don't understand. That's a big gamble to subject the business to.

We'll discuss threat modeling shortly, but one method that can be useful is to consider an application's most likely *threat profiles*. Think of these as use cases, but from a security perspective: What are the most likely avenues of attack your application absolutely must be able to defend itself against?

For example, we often refer to the "coffee shop attack" when discussing web application security. That is, consider an attacker on an open Wi-Fi network (say, in a coffee shop) using a network sniffing tool to eavesdrop on all the network traffic traversing the Wi-Fi. Now, consider one of your application's users using your application in that same coffee shop. Can your application withstand that level of scrutiny, or does it hemorrhage vital data such as user credentials, session tokens, or sensitive user data?

Similarly, when a mobile application is designed for a smartphone or tablet computer, the most likely risk a user faces is from data left behind on a lost or stolen device. If your application is on that lost/stolen device, what information could an attacker find on the device using forensic tools? Does your mobile application store sensitive data locally on the mobile device?

In designing your security mechanism, keeping a handful of these threat profiles in mind is a healthy thing to do. Of course, the threat profiles need to be specific to your application's architecture, but it's generally not too difficult to find data on attacks against similar architectures.

Implementation Considerations

There are many aspects of design that directly overlap with implementation. One aspect, in particular, is in designing where to place various security implementations in an application's design. For example, the next chapter discusses input validation extensively. If we were to implement input validation without any regard for our application's design, we would be quite likely to end up with a functioning but unmaintainable mountain of junk.

The reason for this is that it is entirely feasible to build input validation code at just about any layer of abstraction within an application. Further, if input validation is implemented "on the fly," there is a tendency to include such things as regular expressions throughout the code base. This is what can easily result in code that is basically impossible to maintain over time.

So, despite the fact that something like input validation—which essentially does nothing to enhance the functional features of an application—is generally viewed as an implementation detail, it is important to design it carefully into one's security architecture. In the case of input validation code, it should be centralized in implementation and features, as appropriate. If there are multiple distinct functional layers in an application (for example, Web UI, XML processing, LDAP access, and so on), validation can be centralized on a *per-layer* basis—but never scattered around randomly. The same thing goes for access control, which would be quite impossible to slap on after the fact, if it was not properly designed into the product from its conception. And in each of these cases, care should be taken to retain control on the server side of the application, regardless of what validation or access control might be done on the client.

These design considerations can have an enormous impact as well. Again citing the case of input validation, if we implement our input validation at the application's presentation layer, it will largely preclude us from being able to implement tier 2 or tier 3 security features into the application, simply because of the lack of features available to us in the presentation layer.

We'll discuss detailed examples of this in the next chapter, but for now, let's at least be entirely cognizant that our design decisions need to have a firm footing in the reality of our intended implementations.

Design Review Practices

The common denominator in all development methodologies is to start with a clear understanding of the proposed design of the product. Although we've seen design documents span an enormous spectrum of detail, there is simply no substitute to really knowing and understanding the product's design before proceeding.

Irrespective of your development methodology, you should have a fundamental description of the application and its components. A diagram visualizing all the components is a good starting point (see Figure 3.1). It should include all the physical as well as logical components of the application, at a bare minimum.

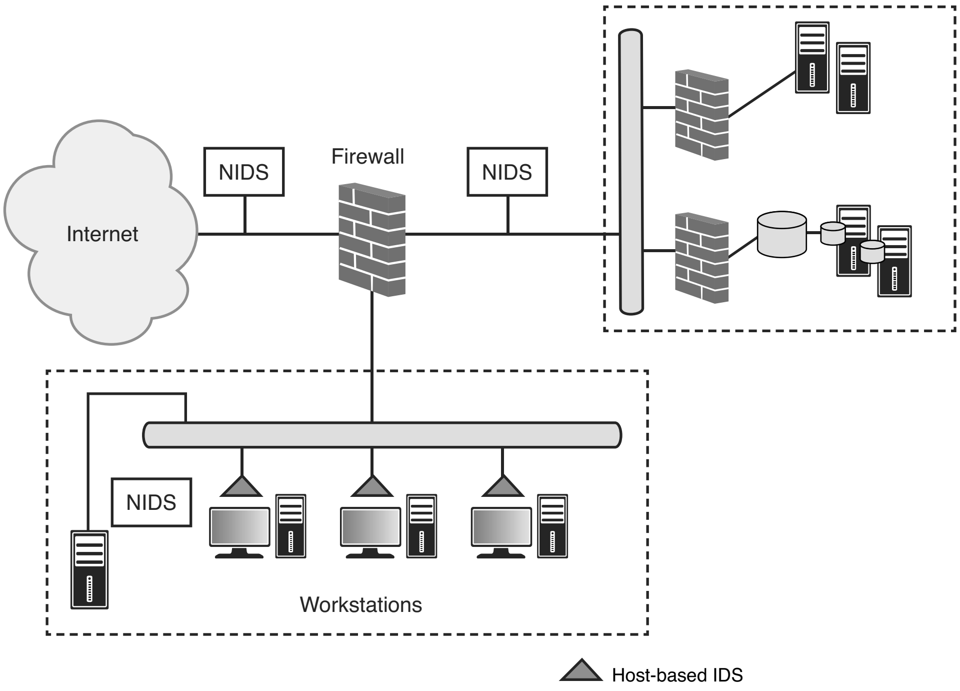


Figure 3.1 Diagram of a typical enterprise application design, top view.

Other useful things to include in design documentation include the following:

- **Components**

  Each component of an application should be described, at least at a top level. What is it called? What is its primary functionality? What security requirements does it have? Who should access it?

- **Data elements**

  What data does the application handle? What are the sensitivities of the data? Who should be allowed to view the data? Who should be allowed to alter the data?

- **Data interfaces**

  How will data be exchanged through the various components of the application? What network protocols will be used? What format(s) will the data be exchanged in?

- **Bootstrap process**

  How is the overall system instantiated? What security assumptions are made during the bootstrapping? How do the application's components mutually authenticate?

- **Shutdown process**

  On shutdown, what is done with system resources, such as open files, temporary file space, and encryption keys? What residue is left behind and what is cleaned up?

- **Logging process**

  This is the process and components for event logging, to include debugging and security logs.

- **Failure process**

  This includes processes for handling system failures, from relatively simple to catastrophic.

- **Update/patch process**

  This consists of the processes for installing component updates, at various levels and components within the application.

The preceding description is highly simplistic, but at the same time it can seem pretty daunting. If you really take the time to do each of these steps in detail, threat modeling a typical application can be an enormously time-consuming process, which can be time and cost prohibitive for many enterprises. As such, this outline is intended merely to give you an overview of what is involved in the threat modeling process. What's important from our standpoint is how to put any of this into practice.

We've had success at breaking down this threat modeling approach into a more simple process, based on the work of many people in the software security field today.

After you have a clear picture of how the application will work, it's useful to break the design into individual operational security zones, particularly for a distributed application with multiple servers and one or more clients.

Next, for each zone and each element in each zone, articulate (preferably in a table format) each of the following: who, what, how, impact, and mitigation.

- *Who?*

  Who can access the zone or element? Not just the registered, authorized users, but who could access the item in general. If it is (say) an application on a mobile device, consider the legitimate user, a user accessing a lost or stolen device, and so on. Try to be reasonably comprehensive in this step. What you're doing is articulating the *threat agents.*

- *What?*

  What can each threat agent do to potentially harm the system? Can the threat agents steal a hard drive from a server? Access a table in a SQL database? Masquerade as a legitimate user and get to sensitive data inside the application? And so forth.

- *How?*

  How can the threat agents carry out each of the things in the *what* list?

- *Impact?*

  If the attack is successful, what is its impact on the business? What are the direct costs as well as the indirect costs? In what ways is the business's reputation tarnished?

- *Mitigation?*

  For any given attack, what mitigation options are available? How much would they cost (in "low," "medium," and "high" terms if more quantifiable data are not available)?

Assuming you've assembled the right team of design reviewers, it's quite likely this approach will result in some useful and meaningful discussions about the application's design.

You should try to consider as many aspects of the application's design as feasible. These reviews can vary from taking a few hours to taking many days, depending on an application's complexity, and how much detailed analysis is expected.

So it needn't be so difficult to do. And we've certainly found the payoffs to justify the time spent. You might find Cigital's approach to be pretty similar in many ways to the threat modeling process we've just described.

It's Already Designed

Throughout this chapter—and indeed in much of this book—we have assumed you would be largely working on new projects, rather than assessing or continuing prior work efforts. Well, that can be a rather unrealistic assumption these days. So let's take some time to dive a bit deeper into what sorts of design security activities can be reasonably accomplished even after a system is in production.

One immediate disadvantage we see here is that ex post facto design changes tend to be costly,[5] so you will doubtlessly encounter more than a little initial resistance in trying to make any substantive changes to the design. This is particularly true for projects that are widely deployed and not (just) run from centralized data centers. This means that any design changes that might come out of an ex post facto review need to be rigorously researched and cost-justified in order to be successful. We believe that this inertia is more than likely to result in workarounds and compromises more often than it results in real design changes, which will no doubt present their own challenges over time. Moreover, some fundamental security issues in legacy applications simply cannot be accomplished without complete rewrite of the application. This might happen due to inherently insecure application architectural choices or simply because of underlying platform or language shortcomings—more on that in Chapter 8, "Maintaining Software Securely." But we should still press on.

Another common hurdle to clear is in "simply" finding the design itself. Many software products and systems are built and deployed without any sign of design documentation. Often, the design "documentation" lives in the brains of the people who designed the system in the first place, and quite possibly those people have moved on to other jobs, perhaps in different companies. We feel that documenting a system's design, even if it is done only ex post facto, is in and of itself a huge benefit that will find value over time even if no changes are made by way of a security assessment. Indeed, two of your authors have amassed considerable experience doing just this at a major corporation. A further pitfall of a system that lacks a clearly documented design is that seemingly innocuous changes can often lead to spectacular failures. It is vital for the entire team to have a clear, comprehensive, and deep understanding of the underlying system.

So enough with the impediments; let's look at how to do design review of a deployed system. Here are some general steps to consider, along with some tips and recommendations in accomplishing them.

1. Document the design.

   Start by looking at whatever existing design documentation is available, of course. Depending on your organization's software development process, this could range from nonexistent to voluminous. The important thing is to "get your head around" the design and thoroughly understand what the software is doing. In our experience, this is best accomplished through a combination of documentation, visualization, and human interviews. So start by sitting down, turning off all interruptions, and studying whatever documentation you have. Care should be taken to validate that the derived design accurately depicts the running system.

   Then, if at all possible, seek out the design team and spend some quality time with them and a large whiteboard. Draw the top-level design in the form of the major software elements and dependencies, their functions, communications channels, data, and so on. Next, overlay a state diagram onto that design, and do your best to understand the software's different states of operation, including initialization, steady-state operation, exception modes, shutdown, backup, availability features, and so on. Take this discussion as deep as you're able to. Look, for example, at each component interconnection and ask how it is identified, authenticated, protected, and so on, as well as what network protocols and data types are being passed. Be sure to document any assumptions at this point
   as well.

   Perhaps most important during this whiteboard exercise, ask questions. Assume nothing. Look for unanticipated failure states. And document all the answers you get. You might well find that your own documentation has details in it that surpass the existing design documentation, such as it might be.

   If you're lucky, much of this will already be in place. But even if that is the case, your priority right now has to be attaining a high degree of understanding of what the documentation says. So even in that fortunate case of having ample documentation, it is still worthwhile to interview the design team and have them explain things in their own words. This will help galvanize your understanding of the system, as well as potentially point out areas of ambiguity and downright errors that might exist in the documentation itself.

2. Perform threat modeling.

   After you are confident that you have a deep and thorough understanding of the documentation, go through the threat modeling process as we've described. You might well have collected ample fodder for this in documenting the design (or studying the extant documentation).

3. Assess the risks and costs.

   Either directly during the threat modeling process vor separately, it's vital to assess the risks and prioritize them.

4. Decide on a remediation strategy.

   The toughest part at this point could well be deciding what the right threshold of risk is for the system you're reviewing. The biggest difference in doing this now versus during the initial product development is that remediation costs are likely to be substantially higher. And not just the direct costs, but the indirect ones. For example, how will deploying a new design inconvenience your customers? How will it affect backward compatibility? The answers to these questions are extremely important.

   The remediation strategy, then, must take all these answers into account, in addition to the normal business impact justifications. For example, a fairly low-impact design defect might well get remediated because its costs are relatively low, whereas a higher impact problem might be delayed until a major release cycle because the costs do not justify the value.

5. Fix the (justified) problems.

   Any issues found that meet your remediation strategy should now be fixed. And, as you might well imagine, it's never just as simple as coding a fix and then checking off the issue as being finished. Since the issue is now, by definition, an important one, it becomes important to dive deeply into the issue.

6. Verify the fixes.

   It's of course not enough to say something is fixed; you have to prove it as well. Especially for security weaknesses, whenever possible you should build test cases that explicitly test for each weakness.

Try also to consider similar classes of weaknesses that might exist elsewhere in the system.

7. Lather, rinse, repeat.

   Essentially, keep iterating through this until you're finished. Mind you, defining that end point isn't easy. What's important is to give each application a level of scrutiny that its value warrants, and to keep at it until that level has been met.

With luck, you won't have unearthed any truly catastrophic design flaws during this process on an application that is already deployed. And if you did, hopefully you found it before your adversaries did. As we said, major design flaws can be hugely costly to fix after an application is deployed.

We should add that it has been our experience that applications already deployed rarely get ex post facto design review scrutiny. That sort of review would usually happen only for the most security-critical software and, even then, usually only if other security flaws have been discovered. Another factor that makes this difficult is when staff is reassigned to work on other projects after any given project has drawn to completion.

Deployment and Operations Planning

Think it's too early in the game to start considering how the system is going to be deployed and operated? Guess again. We'll come back to this in a lot more detail in Chapter 7, "Operating Software Securely," but for now it is important for us to be sure to understand how our application is going to get deployed and operated. For example, how are our application components going to integrate into the intrusion detection infrastructure in the organization?

The Design Review

Today is the first meeting of the newly minted Product Security Team (PST) for the ongoing development of version 2.0 of Peabody's Wandering Medic (WM) app. The first step is to decide on some of the basics and what things can reasonably be done, despite the fact that the development team is already well into the implementation of WM 2.0.

It is agreed that a lightweight design review would be the first course of action. The team would use this opportunity to get to know each other, as well as for everyone to thoroughly "get their heads around" the WM 2.0 design itself. After all, the Peabody team had, to date, only seen the WM apps from the outside, and this is their first deep dive into how the system will work.

So the Peabody CTO starts the process by asking the dev team, apparently without a hint of irony, to produce whatever design documentation they currently have. The dev team nervously grins and proclaims their design to be their source code. The code is modular; it is laid out logically; any first-year computer science graduate should be able to look at the code and intuitively understand the flow of operation. The CTO is less than convinced, and he insists on whiteboarding a top-level design flow of the app and all of its components, including both the mobile client components and the server-side components. The dev team is quite frustrated, but because they really want to see the new merger succeed, they agree.

It takes the dev team and CTO staff a good solid four hours of discussions and whiteboard sketching to put together their visualization of the WM 2.0 design. The whiteboard, although quite cluttered and ugly, now contains the first design view of WM 2.0 (see Figure 3.2). It has been a frustrating process, but not without value.

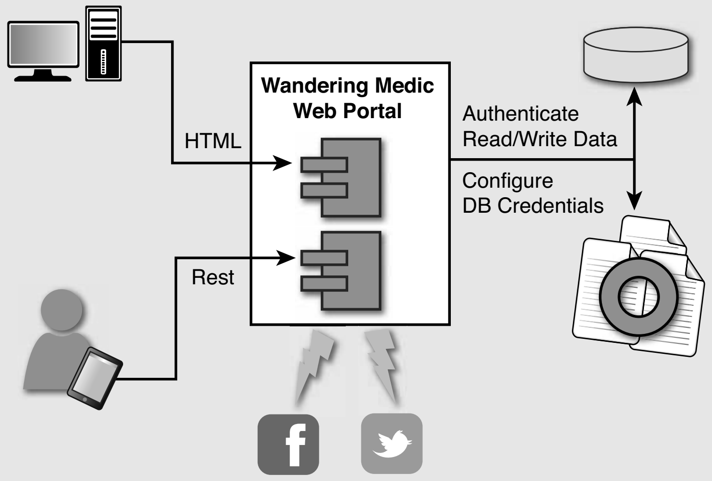


Figure 3.2 The architecture of the Wandering Medic portal.

Most of the frustrations have been because the dev team had differing opinions about how some components of the system would work in the final 2.0 product. The discussions have been lively and heated, but the good news is that the dev team itself has been able to solidify their own understanding of the system they're building. In fact, they even reluctantly agree that the mere process of writing their design—such as it is—on the whiteboard has helped them. They stop short of saying that the four hours in the conference room were better than four hours of coding, though.

Still, drawing the design is only the first step in the design review. The team takes a break until morning, when they'll do a quick design review using a threat modeling approach.

Freshly rested, our PST returns to the conference room to start the second phase of their process, the threat model.

With the design drafted, the CTO suggests dividing it into security zones as a starting point. The functional zones are easy: mobile zone, middleware/controller zone, database zone, authenticator zone. As a starting point, these delineations help the team consider them one at a time.

The first step is now to consider all the threat agents (people, mostly) who have access—legitimate and otherwise—across each zone. To get things rolling, the CTO suggests starting with the mobile zone. Then they will brainstorm all the threat agents they can come up with and then consider each in light of the following details. They must be sure to clearly set the rules for the brainstorming session in advance.

- Doctors

  Clearly, the primary users of the app are the doctors. They have legitimate access to the devices and the app itself. What could motivate a doctor to attack the system? (1) Financial gain—a doctor might look through patient records to brainstorm "legitimate" medical procedures the patient should receive. Far-fetched, but this is a brainstorming exercise.... (2) Privacy violation—a doctor might want to read the records of a VIP patient, out of curiosity, or to give/sell the info. (3) Competition—a doctor might want to read another doctor's patient information to find out how the other doctor is treating a specific ailment. (4) Coverup—a doctor could attempt to remove or alter information to cover up mistakes. The team continues this brainstorming to come up with as many things as they can that could motivate a doctor to attack.

  Now, for each of these potential attacks, the team considers how (technologically) skilled the doctor would need to be in order to be able to succeed.

- Nurses

  Although the primary app users are the doctors, without a doubt nurses will also need to use the app. However, they might well have different motivations for (potentially similar) attacks. (1) Embarrassment—a nurse might enter false data to make a doctor look bad. (The team considers that attack case to be more likely than the other way around.) (2) Many of the same attack cases the team identified for doctors also exist for nurses.

- IT technicians

Certainly, the hospital's own IT staff would have to have access to each mobile device. While discussing this, the team briefly considers the case of "BYOD" (bring your own device) situations. The debate on this topic gets quite heated, with the developers primarily saying that they absolutely want to support BYOD situations. In the end, however, the Peabody CTO prevails and says that a medical enterprise environment will never allow patient data on a BYOD device, so they dismiss that case. Secretly, the CTO hopes that decision doesn't come back to haunt them later.… So what would motivate the hospital's own technicians to attack a system? (1) Sabotage—the techs are disgruntled because of continued budget cutbacks, and they want to teach the big hospital conglomerate a lesson. (2) Sabotage—the techs might want to build a Trojan horse into the app deployment so that the app fails if/when a technician is laid off. (3) Financial gain—the techs know about VIP patients in the hospital, and they could use access to the app to look up private details and then sell that information externally. (4) And again, this list continues. The skills required to do an attack, however, are far more prevalent in our IT technicians than in the doctors and nurses.

- Patients

From time to time, it is quite conceivable that a patient will have access to a doctor or nurse's mobile device. In those situations, what might motivate a patient to attack the app? (1) Alter records—a patient might change his own record to show more/less severe ailments. Perhaps this is financially motivated, to attempt to be charged less for a medical service. (2) Drugs—a patient could try to use a device to add or alter prescriptions, for various reasons. (3) Privacy violation—a patient might attempt to look up someone else's medical records.
Of course, patients' technology skills will span a broad spectrum, so it's best to assume that the patient is very technically astute.

- Doctors' families

Unless the hospital has a policy of not allowing staff to take mobile devices home, the team has to consider what the families could do. Here it might be a good idea to consider some actual attack cases as well as inadvertent attack cases. (1) Games—perhaps it's cliche[as], but it still bears consideration. What happens if/when a doctor's family members try to install games (or other nonsanctioned apps) on a hospital-owned mobile device?

Table 3.2 provides the final summary of potential threat agents that the PST team has come up with after an exhausting deliberation, as well as their potential motivations and expected technical skill levels.

Table 3.2 Threat Agents

Threat Agent	Motivation	Technical Skills
Doctors	Financial gain Privacy violation Competition Coverup	Low
Nurses	Embarrassing doctor Financial gain Privacy violation Coverup	Low
IT Technicians	Sabotaging hospital Sabotaging application Privacy violation	High
Patients	Alter records Financial gain Drug access Privacy violation	High
Doctors' families	Using third-party apps Privacy violation Embarrassing doctor	Low
Thieves of mobile device	Privacy violation Sabotaging hospital	High
Device tech support	Privacy violation Sabotaging hospital Sabotaging application	High

Next, keeping its scope to the mobile zone, the team considers all the following factors. When carrying out this step, the team continues its brainstorming approach. One technique to facilitate this is to use a large whiteboard and construct a matrix (see Table 3.3) with each of the following column headings:

- Target

  Here, the team considers all the targets within the mobile zone. The targets can be technical, data, or process based. For this app, the team considers its primary targets to be patient data (including PII, prescriptions, diagnoses, etc.), medical staff accounts, and encryption keys from back-end applications.

- Vulnerability

To consider all the weakness states of these targets, the team must deeply understand the app and how it functions. In particular, they need to consider all the circumstances where the target data is stored on a mobile device, whether in nonvolatile storage, in volatile storage, or in transit to/from the device.

- Attack

To brainstorm the potential attacks, we need to deeply understand the application's architecture—and force some vital decisions—and understand how things will be implemented. For the team's first shot at doing a threat model, they agree to hold off on several of the technology decisions and then revisit them later. For example, one attack they consider is to gain access to any local data stores via their SQLite (or whatever database the implementation team uses) files, which can be trivially done on iOS or Android versions of their app. They also consider carefully whether the access is read-only or read/write.

In fact, many of the issues the team will consider at this point will slowly be updated as the development process continues.

- Impact

There are two types of impacts any vulnerability will have: technical and business. Whereas the development team understandably concentrates on the technical side of this activity, business stakeholders become worried about business impact, so they very strongly push the team to include business considerations in the impact analysis.

- Remediation

This is where the technical world really clashes with the business reality. If the vulnerability remediation requires completely rearchitecturing the solution, management gets quite impatient. This is where the impact analysis is really utilized, because the team and business folks will have to evaluate the *risks* of additional costs and delays incurred while fixing security issues versus not doing it and potentially being hit with breaches, data losses, and similar niceties. None of which Peabody wants to happen, of course—and neither does its CTO want to reimplement the portal.

Table 3.3 Matrix

Target	Vulnerability	Attack	Impact	Remediation
Patient's data	Stored on device Stored on SD Transmitted in clear			
Encryption keys	Stored on device Transmitted in clear			
Staff accounts	Stored on device			

Summing Up

Whatever your application, and no matter what security tier you think it falls into, taking the time to think it through to the end before you start coding is essential. There is a reason many of the masters of chess and the Oriental board game Go train themselves to sit on their hands when considering their next move. Contemplation, even with a move clock ticking, pays off.

Perhaps you use a formal development life cycle. Maybe you've formed a project security team, or will just kick around ideas with friends. (Or will you, lacking a collaborator, explain your technical problems to an empty chair? It has worked for us.) Whatever your process, we hope the structure and tips in this chapter ease the "take it apart; good, now put it back together" thinking essential to "confluent" design.

Ready, set, go! It's Implementation time.

Endnotes

1. Gary McGraw, *Software Security: Building Security In* (Boston: Addison Wesley Professional, 2006).

2. Michael Howard and Steve Lipner, *The Security Development Lifecycle* (Redmond, Washington: Microsoft Press, 2006).

3. For details, see "SEC Interpretation, Electronic Storage of Broker-Dealer Records," Securities and Exchange Commission, www.sec.gov/rules/interp/34-47806.htm.

4. McGraw, 2006.

5. Barry Boehm, *Software Engineering Economics* (Upper Saddle River: Prentice Hall, 1981).

This page intentionally left blank

This page intentionally left blank

Index