Web Application Security Engineering

Integrating security throughout the life cycle can improve overall Web application security. With a detailed review of the steps involved in applying security-specific activities throughout the software development life cycle, the author walks practitioners through effective, efficient application design, development, and testing.

J.D. Meier
Microsoft

Once upon a time, not long ago, I was a fly on the wall during a security review for a major application rolling out into production. One of the security reviewers asked the development team a seemingly innocent question: "Are you encrypting the data in the cookies?"

Apparently, the development team wasn't encrypting the data in the cookies. Equally apparent, the security reviewers considered this important. The review quickly unfolded into a stream of subsequent questions that turned into a fast-paced ping-pong match between the development team and the security reviewers. The security reviewers challenged the development team's assumptions, and the development team challenged the rationale of their security reviewers.

I'd like to say it had a happy ending, but it didn't. The application was blocked from rolling into production, and the security reviewers made some new enemies. I couldn't help but think that there had to be a better way. Why was the development team finding out so late, why were there so many unchallenged assumptions this late in the cycle, and why was there such a gap between the security experts and the development team?

With this article, I share a way to improve Web application security by integrating security throughout the life cycle. The ideas I present here are based on empirical evidence from consulting with hundreds of customers—real-world scenarios with real project constraints and security concerns—across a variety of scenarios and putting into practice the security techniques that the experts know. The result is an approach that has evolved and refined itself over time.

Proven path

Is there a way to improve Web application security in a repeatable way? Is there a way to connect practitioners to proven software security expertise? Is there an approach that can scale up or down depending on the size of your project or team? Is there a set of techniques that can be incrementally adopted? Absolutely.

To truly appreciate success, you need to understand and appreciate failure. In this article, I first recap a set of approaches that don't work and then walk through a more successful approach that I've evolved over time. As with any approach, scenarios and context matter. However, this approach is easily tailored and can stretch to fit a variety of scenarios.

Approaches that don't work

If it's not broken, then don't fix it. The problem is you might have an approach that isn't working or that isn't as efficient as it could be, but you might not know it yet. Let's take a quick look at some broken approaches and get to the bottom of why they fail. If you understand why they fail, you can then take a look at your own approach and see what, if anything, you need to change.

Bolt-on approach

The bolt-on approach's philosophy is to "make it work, and then make it right." This is probably the most common approach to security that I see, and it almost always results in failure or at least inefficiency. This approach results in a development process that ignores security until the end, which is usually the testing phase, and then tries to make up for mistakes made earlier in the development cycle.

The assumption is that you can bolt on just enough security at the end to get the job done. Although the bolt-on approach is a common practice, the prevalence of security issues you can find in Web applications that use this approach is not.

The real weakness in the bolt-on approach is that some of the more important design decisions that impact an application's security have a cascading effect on the rest of the application's design. If you've made a poor decision early in the design, you'll face some unpleasant choices later. You can either cut corners, further degrading security, or you can extend your application's development and miss project deadlines. If you make your most important security design decisions at the end, how can you be confident you've implemented and tested your application to meet your objectives?

Do-it-all-up-front approach

The opposite of the bolt-on approach is the do-it-all-up-front approach. In this case, you attempt to address all your potential security concerns up front. It has two typical failure scenarios:

- You get overwhelmed, frustrated, and give up.
- You feel like you've covered all your bases and then don't touch security again until you see your first vulnerability in the released product.

Although considering security up front is a wise choice, you can't expect to do it all at once. For one thing, you can't expect to know everything up front. More important, this approach doesn't recognize that you'll make decisions throughout the application development life cycle that affect security.

Big-bang approach

Similar to the do-it-all-up-front approach, the big-bang approach is where you depend on a single big effort, technique, or activity to produce all your security results. Depending on where you drop your hammer, you can certainly accomplish some security benefits, and some security effort is certainly better than none. However, similar to the do-it-all-up-front approach, a small set of focused activities will give you more value for the same effort as a single big bang.

The typical scenario is a shop that ignores security (or pays it less than the optimal amount of attention) until the test phase. Then, it spends a lot of time and money on a security test pass that tells the shop all the things that went wrong. The developers then face a host of hard decisions on what to fix versus what to leave and must attempt to patch an architecture that wasn't designed properly for security in the first place.

Buckshot approach

In the buckshot approach, you try a bunch of security techniques on your application, hoping you somehow manage to hit the right target. In this case, it's common to hear, "we're secure, we have a firewall," or "we're secure, we're using SSL." This approach's hallmark is that you don't know what your target is, and the effort expended is misguided and often without clear benefit. Beware the security zealot who's quick to apply everything in his or her tool belt without knowing the actual issue to defend against. More security doesn't necessarily equate to good security. In fact, you might be trading off usability, maintainability, or performance, without improving security at all.

Essentially, you can't get the security you want without a specific target. Firing all over the place (even with good weapons) isn't likely to get a specific result. Sure, you'll kill stuff, but who knows what.

All-or-nothing approach

With the all-or-nothing approach, maybe you used to do nothing to address security and now you do it all. Overreacting to a previous failure is one reason you might see this type of switch. Another is someone truly trying to do the best he or she can to solve a real problem but biting off more than he or she can chew.

Although it can be a noble effort, this is usually a path to disaster. Multiple factors, aside from your own experience, impact your success, including your organization's maturity and the buy in from team members. Injecting a dramatic change might help gain initial momentum, but if you take on too much at once, you might not create a lasting approach and will eventually fall back to doing nothing.

The approach that works

So if bolting security on to your application or taking a buckshot approach isn't producing effective results, what works? The answer is a life-cycle approach, or "baking" security into your application's life cycle. By leveraging it in the development life cycle, you address security throughout development instead of up front, after the fact, or, worse, randomly. Baking security into the life cycle is also a practical way to balance security with other quality attributes, such as performance, flexibility, and usability.

How do you do this? To answer that, we need to explore which development activities produce the most effective results.

High return on investment

How do you know which techniques to use to shape your software throughout its life cycle? Start with the high return-on-investment (ROI) activities as a baseline set. You can always supplement or modify them for your scenario.

Most development shops have some variations of the following activities, independent of any security focus:

- design guidelines,
- architecture and design review,
- code review,
- testing, and
- deployment review.

Identifying design guidelines involves putting together a set of recommendations for the development team. These recommendations address key engineering decisions (such as exception management) and include input from software vendor recommendations, corporate policies, industry practices, and patterns. It's a high ROI activity because it helps create the project's scaffolding.

Architecture and design review is an exercise in evaluating the design against functional requirements, nonfunctional requirements, technological requirements, and constraints. It can be as simple as a whiteboard sketch, or it can involve multiple documents, diagrams, and presentations. An architecture and design review is most valuable when it's performed early enough to help shape your design. It's less valuable when it's performed so late that it's only function is to find "do overs."

Code review is a practical and effective way to find quality issues. Although you can find some issues through static analysis tools, the advantage of a manual code review is contextual analysis—for example, you know the usage scenario and likely conditions.

Testing is executable feedback for your software: it either works or it doesn't. Whether it works "right" is a bit trickier to establish. The ideal case is the one in which you establish an executable contract between your code and requirements, including functional, nonfunctional, and constraints. It also helps to know what you're looking for so you can test against it. Although testing is usually optimized around functional behavior, you can tune it for quality attributes depending on your ability to set measurable objectives and define what "good" looks like.

Deployment review is an evaluation of your application as it's deployed to your infrastructure—where the rubber meets the road. This is where you evaluate the impact of configuration settings against runtime behavior. A deployment review can be your last actionable checkpoint before going into production. Preferably, you have a working build of the software early in its development cycle so that early deployment reviews can help reduce surprises going into production.

These activities offer a high ROI because, if applied properly, they directly affect the software's shape throughout the process. I've seen these activities transform my customers' results for the better, and they provide key feedback and action at high-leverage points in the development process.

(For more information on which activities produce high ROIs, see Steve McConnell's "Business Case for Better Software Practices" at www.stevemcconnell.com/BusinessCase.pdf.)

Security engineering baseline activities

You can create effective security activities based on the high ROI activities I've listed. However, rather than interspersing security in your existing activities, factor it into its own set of activities. This will help optimize your security efforts and create a lean framework for improving your engineering as you learn and respond. To influence software security throughout the development life cycle, you can overlay a set of security-specific activities. Table 1 shows the key security activities in relation to standard development activities.

The following is a set of baseline activities for security engineering:

- objectives,
- threat modeling,
- design guidelines,
- architecture and design review,
- code review,
- testing, and
- deployment review.

These activities complement one another and have a synergistic effect on your software security.

Objectives

To perform any job, you need to know where to start and when you're done. The security objectives activity is an explicit activity that helps you set boundaries and constraints for your software's security so that you can prioritize security efforts and make any necessary trade-offs.

Security is a quality attribute that you must balance with other quality attributes and competing concerns, as well as project constraints. You should identify up front how important security is for your particular project and context. Identifying security objectives helps you identify when you're done from a security perspective.

A starting point might be to state what you don't want to happen with respect to security—for example, you don't want to leak customer data or for one user to see another user's data. For this activity, it's important to first agree what is inside the scope of your information security. It helps to think in terms of the information's confidentially, integrity, and availability (CIA). You next decide which categories of objectives to work with. Using categories such as client data, compliance, quality of service, and intangible assets helps here.

To determine your security objectives, consider the following questions:

Table 1. Standard development activities and security add-ins at each phase.

MAIN ACTIVITIES	CORE SUBACTIVITIES	SECURITY ADD-INS
Planning		
Requirement and analysis	Functional requirements Nonfunctional requirements Technology requirements	Security objectives
Architecture and design	Design guidelines Architecture and design review	Security design guidelines Threat modeling Security architecture and design review
Development	Unit tests Code review Daily builds	Security code review
Testing	Integration testing System testing	Security testing
Deployment	Deployment review	Security deployment review
Maintenance		

- What client data do you need to protect? Does your application have user accounts and passwords, customer account details, or financial history and transaction records?
- Do you have compliance requirements such as a security policy, privacy laws, regulations, and standards?
- Do you have specific quality of service requirements?
- Do you need to protect intangible assets, such as your company's reputation, trade secrets, or intellectual property?

In practice, two techniques can really help identify security objectives. One is to use a roles matrix, which is simply a matrix of the roles in the system, the objects that they act on, and their permissions on those objects. You can use the matrix to easily see what roles shouldn't perform which actions. Another technique is to list the system's functional requirements and then identify corresponding CIA requirements as appropriate for each functional requirement.

The following guidelines can help you identify your own security objectives:

- Use scenarios to set bounds. Are operational requirements out of scope?
- Ask yourself, "What's the worst thing that could happen from a security perspective?" This helps you identify how important security is and where to spend more energy.
- Turn objectives into requirements and constraints.

Conversely, you can avoid common mistakes when identifying security objectives:

- Don't try to figure out all your security objectives up front. What matters is that you identify the important ones that could have a cascading impact on your design.
- Don't create a document that nobody will ever use or look at again. You should create documents that you need for the next step; anything else can be a checklist. If the objectives are important, turn them into action. You're the best judge here, and you know what gets used and what doesn't. Usually, the work item tracking system is the most effective.

Following these do's and don'ts will give you a start in the right direction.

Threat modeling

You need to be familiar with your system's threats so that you know whether your security mechanism is effective or even warranted. Threat modeling, an engineering technique you can use to shape your software design, will help inform and rationalize your key security engineering decisions. In its simplest form, a threat model is an organized representation of relevant threats, attacks, and vulnerabilities to your system.

During design, the most effective threat-modeling technique lets you quickly play out various "what if" scenarios. Long before you implement an expensive countermeasure, you can quickly evaluate whether you're even going down the right path.

You can certainly use threat modeling on a deployed system, but you'll likely find overlap with existing risk assessment activities. Risk assessments typically focus on the business risk, and often have an operational or infrastructure focus, so they're not usually optimized for direct usage by the software engineer.

In practice, I've found the most successful threat-modeling activities partition the infrastructure threat models from the applications threat models. For some

companies, partitioning by network, servers, desktops, and applications helps optimize relevancy and ownership. The key is to identify the important intersections between the application and infrastructure. Over time, the infrastructure threat models tend to be reusable as input for the application threat models.

When you optimize an activity for a specific problem domain, you can pare down the activity while increasing efficiency.

The following guidelines can help you build a threat model:

- Incrementally render your threat model. Model it to identify what you know, what you don't know, and what you need to know next.
- Use a question-driven approach. Whether you're figuring out threats, attacks, or vulnerabilities, asking the right questions can lead you to better answers. You can even seed your questions with lists of questions that experts ask.
- Use a threat model to identify what code to prototype and test. Your threat model will likely reveal high-risk engineering decisions, which are great candidates to prioritize what to prototype or spike.
- Use complimentary techniques. You can use various techniques for finding threats, attacks, and vulnerabilities, such as use-case analysis, dataflow analysis, question-driven approaches, roles matrices, and brainstorming.
- Use pattern-based frames. They're a conduit of insight from expert to practitioner and can significantly help find the low-hanging fruit as well as organize your thinking.
- Partition your threat-modeling activity within a predetermined time period.
- Identify vulnerabilities from your threat model. The most actionable output of a threat-modeling exercise is a list of vulnerabilities to tackle.

Conversely, you can avoid some common mistakes:

- Don't do it all up front.
- Don't use threat modeling as a silver bullet. It's a key activity, but you should use the right tool for the job. Threat modeling doesn't replace a security architecture and design review or a security code review.
- Don't make your threat modeling such a heavy, formal activity that the software development outpaces your model.
- Don't explain away threats and attacks during threat modeling. You can evaluate countermeasures downstream, after you've focused on identifying the issues you're up against.

(For more information on key concepts and a software engineering-focused approach to threat modeling, see "Threat Modeling Web Applications" at http://msdn.com/ThreatModeling.)

Design guidelines

Is your development team using proven software security design principles, practices, and patterns? If you haven't pulled some together, you have less of a chance of success. The security guidelines activity is where you hunt and gather relevant security design advice and organize it in a way that's useful for your development team. This can include vendor recommendations and home-grown experience.

A good guideline, however, will be more than just a description; it will be prescriptive. "Use input validation" is clearly a good idea, for instance, but it isn't a good guideline. Let's turn that nice idea into something we can act on:

- *What to do*: Don't rely on client-side validation. Use it to help reduce the number of round trips to the server, but don't rely on it for security.
- *Why*: Server-side code should perform its own validation because an attacker might bypass your client or shut off your client-side script routines.
- *How*: Use an "allow" approach over "deny." It's unlikely that you can define all the possible variations of what bad input would look like. Instead, define what good input looks like and filter for that.

An effective way to measure a guideline's success is to determine if it drives results and if it's repeatable. What I like about this schema in particular is that it's simple and self-testing. It's easy to spot a blanket recommendation that's missing the rationale or a guideline that doesn't show you how to actually follow it.

Here's what to look for when building your design guidelines:

- Use categories to frame out the problem space and organize the information.
- Use a what to do, why, and how approach. For each guideline you want your development team to follow, distill it down into actionables, with rationale and code examples where possible.
- Use a simple wording scheme such as do, don't, should, shouldn't, must, and mustn't. This helps qualify the guideline.
- Qualify when the guideline is appropriate and when it isn't.

- Make insightful guidelines. This sounds obvious, but it's easy to fall into the trap of documenting what everybody already knows. Spend more attention on the information that's surprising.

Conversely, you can avoid some common mistakes:

- Don't create all the guidelines up front. Focus on the ones that have an immediate impact on compliance, requirements, and any technologies you're evaluating or using.
- Don't reinvent wheels. There's a lot of available security guidance; the key is to find it and turn it into action for your team.
- Don't lock your guidelines up in a document you'll never use. You need to evaluate the best medium for your guidelines by integrating them into your activities and workflow.

By following these simple rules, you can produce a guideline that is actionable, tells you what to do, why you should do it, and how to take action.

Architecture and design review

You probably already do some sort of design review. It's also likely that you're interspersing security into your current design review activity. The problem with this approach is twofold:

- You're overcomplicating your existing design review.
- You're limiting the effectiveness and efficiency of finding security issues.

Don't get me wrong: general design reviews can be effective, and they certainly have their place, but I've never seen them produce the same results as a focused security design review. When you optimize an activity for a specific problem domain, you can pare down the activity while increasing efficiency. The efficiency gain is a byproduct of precision.

Think of precision as filtering out everything that's irrelevant. While you're focused on security, you aren't randomly focused on other quality attributes, except when there's a direct intersection. As an example of where security and performance intersect, if you're deciding on an SSL solution, your performance requirements might tell you whether you can use a software solution or if you need a hardware solution.

What's important about the architecture and design review is that you identify what you know, what you don't know, and what you need to know next. You have to balance whether it's faster to think you're way through a problem and have a theory or to build out a test and evaluate. The nice thing about a security architecture and design review is that you're effectively prioritizing and scoping what tests to build, if you have to.

The most important first step to the review is to start with the end in mind. You don't need to expose every detail, but you must expose what's important for the decisions at hand. Although you can diagram your system in a variety of ways, I find standing at a whiteboard is effective. The funny thing about a whiteboard is that it forces you to tease out just what's important, whether you realize it or not. A good guideline when showing a visual representation of your system is to focus on the logical layers and the intersections with the physical topology. If you can show what's deployed and where, you're well on your way.

The next best things you can expose are the key security-relevant cross-cutting application features, such as authentication, authorization, input and data validation, and so on. Each of these features impacts the application's design and is important for security.

A good way to quickly determine where to focus for maximum security impact is to ask questions that pull out the high-risk decisions. You can quickly litmus test the design by asking some basic questions:

- How do you authenticate callers on the front end, middle-tier, and back-end resources?
- How do you authorize callers on the front end, middle-tier, and back-end resources?
- How do you flow identity? (Will you impersonate the original caller to the back-end resources, or will you use a trusted proxy on the caller's behalf?)
- How do you validate input? Do you centralize your input-handling routines? Or does each developer whip out his or her own bag of tricks?
- How do you make sure you don't leak sensitive data when exceptions occur?
- How do you audit your critical business functions?

These sorts of questions drive where to investigate next. Depending on the answers, you can start to see just how much you should worry about the design.

Here's how to build an architecture and design review:

- Factor security into its own review.
- Time manage your security design review. Because you don't have infinite time, set aside a specific amount of time to help your review to meet priorities.
- Use a question-driven approach.
- Partition your security design review into smaller chunks if appropriate. For example, review a critical piece of functionality or mechanism, such as your input and data validation approach.
- Build design review gates into the everyday workflow. This improves defect yield by making errors more discoverable via review and static analysis.

And here's how to avoid some common mistakes:

- Don't perform the security architecture and design review too late. It's meant to help catch big mistakes earlier rather than later.
- Don't intersperse your security design review with a general review.
- Don't get stuck on the little things. You need to find the big issues that have a cascading impact.

(For more information, see "Security Architecture and Design Review for Web Applications" at http://msdn.microsoft.com/library/en-us/dnnetsec/html/THCMCh05.asp.)

Code review

What better way to find security issues than to look at the source. Security code reviews aren't about determining whether the code works; running the code tells you that. Here, you determine whether the code meets your security objectives. Security code reviews let you apply contextual analysis as you walk the code to determine whether the behavior is an issue, by design; it meets your requirement; or you need to adjust your requirements.

It's important to keep in mind that some security choices are about following building codes or proven practices, while other choices are more about business risk. For example, following a vendor recommendation for a coding practice is closer to a building code, which might take the form of compliance rules, vendor recommendations, or your own policies.

Other choices might be primarily driven by business risk. When you're faced with a few ways to solve a problem, in the absence of a clear coding practice, business risk is usually a driving factor. Many security practices become a combination of business risk and proven practice. At the end of the day, it's your security objectives that help you know where to set the bar and how far to go.

If you were to analyze the metaprocess that security experts use to review code, you would find a common pattern. I call these algorithms for engineering techniques. Security experts use four common steps to perform security code reviews:

1. *Identify security code review objectives.* This is where you establish the review's goals and constraints.
2. *Perform a preliminary scan.* This involves using static analysis to find an initial set of issues and improve your understanding of where you will most likely find issues when you review the code more fully. The key here is to not get lost in the potential noise and false positives, but to see patterns of where to look further.
3. *Review code for security issues.* This is where you review the code thoroughly to find security vulnerabilities. You can use the results of Step 2 to focus your analysis. Here, techniques such as control- and dataflow analysis come into play.
4. *Review for security issues unique to your architecture.* This step is most important if you've implemented a custom security mechanism or any feature designed specifically to handle known security issues.

Although static analysis is helpful for finding patterns of where to look, contextual analysis and knowing your security objectives are key to knowing where to start, how to proceed, and when you're done.

Here's what to look for when starting a code review:

- Budget your time, so that you can prioritize the right activities, even over multiple sessions.
- Set objectives, but don't get stuck while creating them. Objectives help by setting boundaries and constraints, but they aren't an end unto themselves, so refine them as you go.
- Use a personal checklist. Figure out your own pattern of mistakes and use a checklist to help eliminate them.

Conversely, you can avoid a common mistake: don't just tack a security code review onto your regular code review. It's far more powerful, effective, and lean as its own activity. (For more information, see "How To: Perform Security Code Review for Managed Code (Baseline Activity)" at http://msdn.microsoft.com/library/en-us/dnpag2/html/paght000027.asp and the companion "Security Question List: ASP.NET 2.0" at http://msdn.microsoft.com/library/en-us/dnpag2/html/pagquestionlist0001.asp.)

Testing

Talk about security testing usually focuses on penetration testing. This could be white-box testing, in which you know the internal workings, or black-box testing, in which you don't. Black-box testing usually involves testing boundary values, injecting malicious input, and testing known exploits. There's growing evidence to suggest that white-box testing is far more effective at finding security issues. In either case, penetration testing isn't a silver bullet.

One complaint I often hear is that security testing is unbounded, and that it's tough to tell when you're done. If we look a littler closer, there's usually a root-level cause such as not having a threat model.

A lot of testing is requirements-based because requirements help define an application's scope. The problem is that requirements are more about functionality than qualifiers (such as what "good enough" security looks like). With performance, you can measure response time, throughput, and resource utilization. With security, it's difficult to measure confidentiality, integrity, and availability. The bar for knowing where to start, how to

proceed, and when you're done goes back to your security objectives.

The most effective, efficient security testing revolves around your security objectives and threat model. This doesn't dismiss possible benefits from exploratory testing or discount penetration testing. In fact, it's an industry practice to get a third-party to assess your software security when the stakes are high, which often includes penetration testing, both white and black box.

Deployment review

Deployment is when your application meets the infrastructure, so deployment review is a great opportunity to assess the final end game. Hopefully, production isn't the first time you introduce your application to your deployment scenario. In practice, this often seems to be the case because I continue to hear horror stories in which an application roll out went bad or got blocked for reasons that an earlier deployment review would have caught.

An application can behave differently during development and production scenarios for various reasons. Sometimes the developer doesn't know the infrastructure constraints. Sometimes the software violates some corporate policy that got lost along the way. Sometimes the application is built and tested with a privileged account but breaks when running under a least privilege service account in production. Sometimes the application is built and tested on wide-open machines but breaks when running on a locked down server in production.

Your goal with testing deployment should be to eliminate as many surprises as possible, as early as possible. You should at least have a staging environment before rolling out to production. In the ideal case, you have build cycles earlier in the development in which you can evaluate your software's runtime behavior. Code that runs one way on a developer's machine might run differently in the production case—it's the norm versus the exception. Software that runs as your security context during development, often with the Web server and database on the same machine, might not play that well when running in production. Usually, there's plenty of moving parts with configuration options that can affect how your software behaves in production.

If you have a lot of knobs, switches, and desired settings or states, categories can help. A flat list of settings can be overwhelming, and it's tough to rationalize conflicting lists, particularly from different sources with completely different lists. I like to organize the options by actionable categories. You might name a category "Accounts," for example, and then identify the end state, based on principles:

- accounts are least privilege,
- strong password policies are enforced, and
- separation of privileges is enforced.

You can then quickly normalize compliance rules, infrastructure constraints, vendor recommendations, and so on. This also helps make checking more repeatable during deployment reviews.

Here's what to look for when building a deployment review: essentially, you need to keep your checklists updated. Over time, the landscape, compliance, and infrastructure changes, so your checklists need to keep up.

Conversely, you can avoid a common mistake: don't depend solely on your checklists. They're a great way to catch low-hanging fruit, but they don't replace thoughtful, contextual analysis. (For more information, see the patterns and practices "Security Deployment Review Index" at http://msdn.microsoft.com/library/en-us/dnpag2/html/SecurityDeploymentReviewIndex.asp.)

Making security activities more effective

When it comes to improving security, the primary advantage security-specific activities have over more general activities is focus. They let you put the right amount of effort on security in the right places, as well as optimize your techniques.

That's a first step in the right direction, but there's more. Using categories and context precision help you further focus and produce more effective results.

Web Application Security Frame

Patterns of problems bite us again and again. The Web Application Security Frame (http://msdn.microsoft.com/library/en-us/dnpag2/html/TMWAcheatsheet.asp) is a set of categories you can use to organize recurring issues. It improves the effectiveness of the security activities I've outlined in this article by helping you leverage expertise in an organized fashion while reducing information overload. If you know what to look for, you have a much better chance of finding it and, even better, finding it efficiently.

Here are the categories in the Web Application Security Frame:

Don't depend solely on your checklists. They're a great way to catch low-hanging fruit, but they don't replace thoughtful analysis.

- input and data validation,
- authentication,
- authorization,
- configuration management,

Additional resources

For more information on the ideas I presented in the main text, see the following resources:

- My blog: http://blogs.msdn.com/jmeier/
- Channel9 Security Wiki: http://Channel9.msdn.com/Security/
- Patterns and practices "Security Engineering," http://msdn.com/SecurityEngineering
- Patterns and practices "Threat Modeling," http://msdn.com/ThreatModeling
- Patterns and practices "Security Guidance Index," http://msdn.com/SecurityGuidance

- cryptography,
- sensitive data,
- exception management,
- session management, and
- auditing and logging.

These sets of categories are repeatable and relatively stable over time because they address root-cause issues versus one-off problems.

Let's take a closer look at input and data validation (or a lack of), which is a common Web application feature. It's actionable, and it shapes your design. From a principle standpoint, you shouldn't trust client input. From a practice standpoint, you know that it's better to define good input and filter for that rather than try to define all the variations of bad input. This category encapsulates the information in a distilled and relevant way.

(For more information, see "Cheat Sheet: Web Application Security Frame" at http://msdn.microsoft.com/library/en-us/dnpag2/html/TMWAcheatsheet.asp.)

Context Precision

A Web application isn't a component, desktop application, or Web service. If I gave you an approach to threat model a Web application, you could probably stretch the rubber band to fit Web services, too. You could probably even bend it to work for components or mobile applications. The problem is that type and scenario matter and can sharply influence your technique. If you generalize the technique, you produce generalized results. If you increase the context and precision, factoring that into your technique, you can deepen your impact.

If I'm threat modeling a Web application, for example, and I know the deployment scenario, I can work my way through it. If I'm threat modeling a reusable component that might be used in a variety of situations, I would start with the top three to five likely deployment scenarios and play those out. Although this sounds obvious, I've seen folks try to model all the possible variations of a component in a single messy model. And I've seen them give up right away because there's just too many. The irony is that three-to-five quick little models usually give you the dominant issues and themes.

Categories for context

Context precision is a phrase I give to the concept of evaluating a given problem's context:

- application type (Web application, Web service, component/library, desktop application, or mobile application),
- deployment scenario (intranet, extranet, or Internet),
- project type (in-house IT, third party, shrink-wrapped software, and so on),
- life cycle (RUP, MSF, XP, Waterfall, Agile, Spiral, and so on).

For application type, you could focus on Customer Relationship Management (CRM) or some other business vertical. I dumb it down to the architecturally significant set that I've seen have an immediate impact on the activity. Although it seems like you could just use the same pattern-based frame for Web applications and Web services, you can create a better one optimized for Web services alone: a Web service involves a proxy, which is a great category for evaluating attacks, vulnerabilities, and countermeasures.

As another example, take input validation. For a Web application, you're likely talking about the HTML stream. For a Web service, we're focused on XML validation. On one hand, you don't want to invent a new technique for every context. Instead, you want to pay attention to the context and ask whether the technique was actually designed for what you're doing. But could you further refine or optimize the technique for your product line or context? Asking these questions sets you down the path of improved software engineering.

Factoring security into its own set of activities is preferable to tacking security onto existing activities. Now that you're looking through a security lens, you can separate concerns and quickly make calls on how important security is and where to prioritize. You can also optimize and refine your techniques. I welcome any comments or feedback on this article. □

J.D. Meier *is a program manager at Microsoft. His professional interests include software engineering and quality attributes. Meier is the author of* Building Secure ASP.NET Applications *(Microsoft Press, 2003),* Improving Web Application Security *(Microsoft Press, 2003), and* Improving .NET Application Performance *(Microsoft Press, 2004). Contact him at MyIEEEmail@hotmail.com.*