Risk Analysis in Software Design

Denis Verdon
Fidelity National Financial

Gary McGraw
Cigital

Risk analysis is often viewed as a "black art"—part fortune telling, part mathematics. Successful risk analysis, however, is nothing more than a business-level decision-support tool: it's a way of gathering the requisite data to make a good judgment call based on knowledge about vulnerabilities, threats, impacts, and probability.

Established risk-analysis methodologies possess distinct advantages and disadvantages, but almost all of them share some good principles as well as limitations when applied to modern software design. What separates a great software risk assessment from a merely mediocre one is its ability to apply classic risk definitions to software design and then generate accurate mitigation requirements. A high-level approach to iterative risk analysis should be deeply integrated throughout the software development life cycle.[1] In case you're keeping track, Figure 1 shows you where we are in our series of articles about software security's place in the software development life cycle.

Traditional terminology

Example risk-analysis methodologies for software usually fall into two basic categories: commercial (including Microsoft's STRIDE, Sun's ACSM/SAR, Insight's CRAMM, and Cigital's SQM) and standards-based (from the National Institute of Standards and Technology's ASSET or the Software Engineering Institute's OCTAVE). An in-depth analysis of all existing methodologies is beyond our scope, but we'll look at basic approaches, common features, strengths, weaknesses, and relative advantages and disadvantages.

As a corpus, "traditional" methodologies are varied and view risk from different perspectives. Examples of basic approaches include

- financial loss methodologies that seek to provide a loss figure to balance against the cost of implementing various controls;
- mathematically derived "risk ratings" that equate risk with arbitrary ratings for threat, probability, and impact; and
- qualitative assessment techniques that base risk assessment on anecdotal or knowledge-driven factors.

Each basic approach has its distinctly different merits, but they almost all share some valuable concepts that should be considered in any risk analysis. We can capture these commonalities in a set of basic definitions:

- The *asset*, or object of the protection efforts, can be a system component, data, or even a complete system.
- *Risk*, the probability that an asset will suffer an event of a given negative impact, is determined from various factors: the ease of executing an attack, the attacker's motivation and resources, a system's existing vulnerabilities, and the cost or impact in a particular business context.
- The *threat*, or danger source, is invariably the danger a malicious agent poses and that agent's motivations (financial gain, prestige, and so on). Threats manifest themselves as direct attacks on system security.
- A *vulnerability* is a defect or weakness in system security procedure, design, implementation, or internal control that an attacker can compromise. It can exist in one or more of the components making up a system, even if those components aren't necessarily involved with security functionality. A given system's vulnerability data are usually compiled from a combination of OS- and application-level vulnerability test results, code reviews, and higher-level architectural reviews. Software vulnerabilities come in two basic flavors: flaws (design-level problems) or bugs (implementation-level problems). Automated scanners tend to focus on bugs, since human expertise is required for uncovering flaws.
- *Countermeasures or safeguards* are the management, operational, and technical controls prescribed for an information system that, taken together, adequately protect the system's confidentiality, integrity, and availability as well as its information. For every risk, a designer can put controls in place that either prevent or (at a minimum) detect the risk when it triggers.
- The *impact* on the organization, were the risk to be realized, can be monetary or tied to reputation, or it might result in the breach of a law, regulation, or contract. With-

1540-7993/04/$20.00 © 2004 IEEE

out a quantification of impact, technical vulnerability is hard to handle—especially when it comes to mitigation activities.
- *Probability* is the likelihood that a given event will be triggered. It is often expressed as a percentile, although in most cases, probability calculation is extremely rough.

Although they start with these basic definitions, risk methodologies usually diverge on how to arrive at specific values. Many methods calculate a nominal value for an information asset, for example, and attempt to determine risk as a function of loss and event probability. Others rely on checklists of threats and vulnerabilities to determine a basic risk measurement.

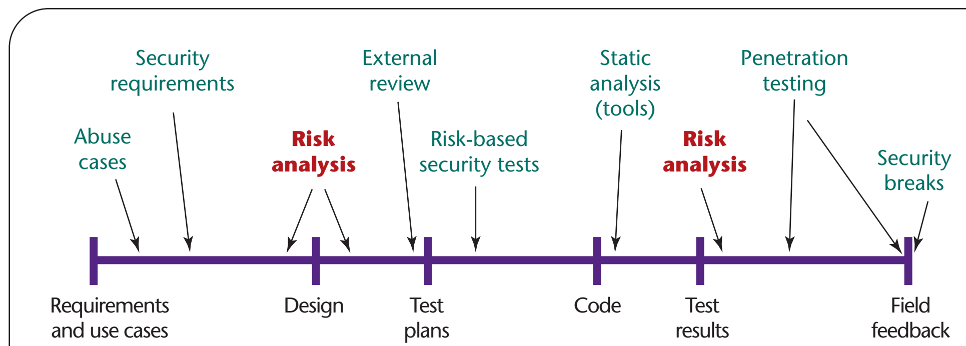

Figure 1. The software development life cycle. Throughout this series, we'll focus on specific parts of the cycle; here, we're examining risk analysis.

Example of a risk calculation

One classic risk-analysis method expresses risk as a financial loss, or annualized loss expectancy, based on the following equation:

$$ALE = SLE \times ARO,$$

where SLE is the single loss expectancy, and ARO is the annualized rate of occurrence (or the predicted frequency of a loss event happening).

Let's consider an Internet-based equities trading application with a vulnerability that could result in unauthorized access (the implication being that unauthorized stock trades can be made). Assume a risk analysis determines that middle- and back-office procedures will catch and negate any malicious transaction such that the loss associated with the event is simply the cost of backing out of the trade. We'll assign a cost of \$150 for any such event, so SLE = \$150. With an ARO of just 100 such events per year, the cost to the company (or ALE) will be \$15,000.

The resulting dollar figure provides no more than a rough yardstick, albeit a useful one, for determining whether to invest in fixing the vulnerability. Of course, for our fictional equities trading company, a \$15,000 annual loss might not be worth getting out of bed for (typically, a proprietary trading company's intraday market risk dwarfs such an annual loss figure).

Other methods take a more qualitative route. In the case of a Web server providing a company's face to the world, the Web site's defacement might be difficult to quantify as a financial loss (although some studies indicate a link simply between security events and negative stock-price movements[2]). In cases in which "intangible assets" are involved (such as reputation), qualitative risk assessment might be a more appropriate way to capture the loss.

Regardless of the technique used, most practitioners advocate a return on investment study to determine whether a given countermeasure is cost-effective for achieving the desired security goal. Adding applied cryptography to an application server via native APIs without the aid of dedicated hardware acceleration might be cheap in the short term, for example, but if it results in a significant loss in transaction volume throughput, a better ROI might come from investing up front in crypto acceleration hardware. Interested organizations should adopt the risk-calculation methodology that best reflects their needs.

Common themes

Most risk-analysis process descriptions emphasize identification, ranking, and mitigation as continuous processes and not just a single step to be completed at one stage of the development life cycle. Risk-analysis results and risk categories tie in with both requirements (early in the life cycle) and testing (where developers can use results to define and plan particular tests).

Because it's a specialized subject, risk analysis is not always best performed solely by the design team. Rigorous risk analysis relies heavily on an understanding of business impacts, which requires an understanding of laws and regulations as well as the business model supported by the software. Because developers and designers build up certain assumptions regarding their system and the risks it faces; at a minimum, risk and security specialists should assist in challenging those assumptions against generally accepted best practice. They're in a better position to "assume nothing."

Putting the right people together for an analysis is important: consider the risk team very carefully. Knowledge and experience cannot be overemphasized because risk analysis is not a science, and broad knowledge of vulnerabilities, bugs, flaws, and threats is a critical success factor.

A prototypical analysis involves several major activities that often include several basic substeps:

- Learn as much as possible about the analysis target (substeps include

Cigital's approach

Figure A illustrates Cigital's continuous risk-analysis process, which loops constantly and at many levels of description through several phases. In Cigital's approach, business goals determine risks, risks drive methods, methods yield measurement, measurement drives decision support, and decision support drives fix/rework and application quality.

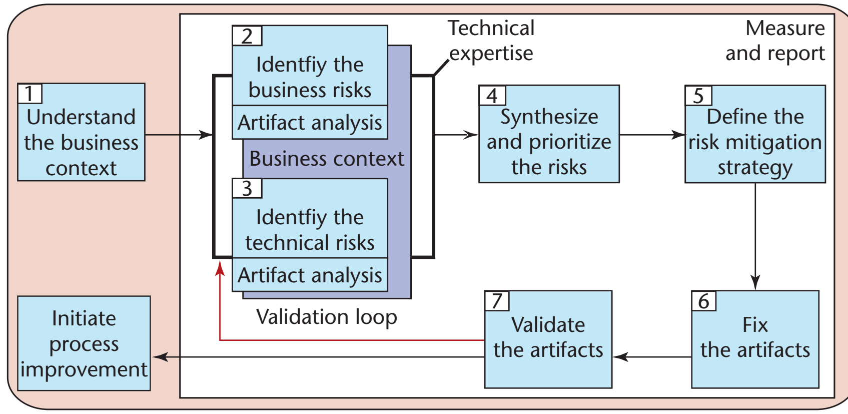

Figure A. Cigital's risk-management framework. Many aspects of frameworks such as this can be automated—for example, risk storage, business risk to technical risk mapping, and the display of status over time.

reading and understanding specifications, architecture documents, and other design materials; discussing and brainstorming with the group; determining system boundary and data sensitivity/criticality; playing with the software if it exists in an executable form; studying the code and other software artifacts; and identifying threats and agreeing on relevant sources of threat).

- Discuss security issues surrounding the software (substeps include arguing about how the product works and determining areas of disagreement; identifying possible vulnerabilities, sometimes by using tools or lists of common vulnerabilities; mapping out exploits and discussing possible fixes; and gaining an understanding of current and planned security controls).
- Determine the probability of compromise (substeps include mapping out attack scenarios for vulnerability exploitation and balancing controls against threat capacity to determine likelihood).
- Perform impact analysis (substeps include determining the impact on asset and business goals and considering the impact on security).
- Rank risks.
- Develop a mitigation strategy (a substep is recommending countermeasures to mitigate risks).
- Report findings (substeps include carefully describing major and minor risks while paying close attention to impact, and providing basic information about where to spend limited mitigation resources).

The sidebar on Cigital's solution shows one commercial example that follows this basic approach.

Knowledge Requirement

Design-level analysis is knowledge intensive. Microsoft's STRIDE model, for example, involves the understanding and application of several threat categories during analysis.[3] Similarly, Cigital's SQM approach uses attack patterns[4] and exploit graphs to understand attack resistance, knowledge of design principles for ambiguity analysis,[5] and knowledge regarding commonly used frameworks (.NET and J2EE being two examples) and software components.

A central activity in design-level risk analysis is to build up a consistent view of the target system at a reasonably high level. The idea is to see the forest, not get lost in the trees. The most appropriate level for this description is the typical "white board" view of boxes and arrows describing the interaction of various critical design components. The nature of software systems leads many developers and analysts to assume (incorrectly) that a code-level description of software is sufficient for spotting design problems. Although this might occasionally be true, it does not generally hold. Extreme programming's claim that "the code is the design" represents one radical end of this approach. Without a white-board level of description, an architectural risk analysis is likely to overlook important risks related to flaws.

Risk Analysis and Requirements

Previous articles in this series consider security requirements definitions and discuss abuse cases as a method for generating requirements. In the purest sense, risk analysis begins at this point: design requirements should take into account the risks you're trying to counter. Let's look at three approaches to interjecting a risk-based philosophy into the requirements phase (note that the requirements systems based on UML tend to focus more attention on security functionality than they do on misuse and abuse cases):

- SecureUML (www.informatik.uni-freiburg.de/~tolo/pubs/secuml_uml2002.pdf) is a met-

hodology for modeling access-control policies and their integration into model-driven software development. SecureUML is based on role-based access control and models security requirements for well-behaved applications in predictable environments.
- UMLsec (http://www4.in.tum.de/~umlsec/) is an extension to UML that enables the modeling of security-related features such as confidentiality and access control.
- Guttorm Sindre and Andreas Opdahl[6] attempt to model abuse cases as a way of understanding how applications might respond to threats in a less controllable environment; they describe functions that the system should not allow.

A key variable in the risk equation is impact. Business impacts generally boil down into three broad categories:

- federal or state laws and regulations (including the Gramm-Leach-Bliley Act, HIPAA, and the much-cited California Senate Bill 1386);
- financial or commercial considerations (such as revenue protection, control over high-value intellectual property, and preservation of brand and reputation); and
- contractual considerations (including service-level agreements and avoidance of liability).

The first step to risk analysis at the requirements stage is to break down requirements into three simple categories: must haves, important to haves, and nice but unnecessary. Unless you're running an illegal operation, you should always class laws and regulations into the first category—these requirements should be instantly mandatory and not subject to further risk analysis (although an ROI study can help you select the most cost-effective mitigations). If the law requires you to protect private information, for example, this requirement is compulsory and should not be subject to a risk-based decision. Why? Because the government has the power to put you out of business, which is the mother of all risks (if you want to test government regulators on this one, go right ahead).

You're then left with risk impacts—the ones that have as variables potential impact and probability—that must be managed in other ways. Examples of mitigations range from technical protections and controls, to business decisions for living with the risk. At the initial requirements definition stage, you might be able to make some assumptions regarding which controls are necessary.

Evenly applying these simple ideas will put you ahead of most application developers. As you move toward the design and build stages, risk analysis should begin to test your first assumptions from the requirements stage by testing the threats and vulnerabilities inherent in the design.

Limitations

Traditional risk-analysis output is difficult to apply directly to modern software design. Even assuming a high level of confidence in the ability to predict the dollar loss for a given event and performing Monte Carlo distribution analysis of prior events to derive a statistically sound probability distribution for future events, there's still a large gap between an ALE's raw dollar figure (as discussed earlier) and a detailed software security mitigation definition.

A more worrying concern is that traditional risk-analysis techniques do not necessarily provide an easy guide (not to mention an exhaustive list) of all potential vulnerabilities and threats to consider at a component/environment level. This is why a large knowledge base and lots of experience is invaluable.

The thorny knowledge problem arises in part because modern applications, including Web services applications, are designed to span multiple boundaries of trust. The vulnerability of—and threat to—any given component varies with the platform on which that component exists (think C# on a Windows .NET server versus J2EE on Tomcat/Apache/Linux) and the environment in which it lives (think secure DMZ versus directly exposed LAN). However, few traditional methodologies adequately address the contextual variability of risk given changes in the core environment. This is a fatal flaw when considering highly distributed applications or Web services.

In modern frameworks such as .NET and J2EE, security methods exist at almost every layer, yet too many applications today rely on a "reactive" protection infrastructure that only provides protection at the network transport layer. This is too often summed up by saying, "We're secure because we use SSL and implement firewalls," which opens the door to all sorts of problems such as those engendered by port 80 attacks, SQL injection, class spoofing, and method overwriting (to name just a few).

One approach to overcoming these problems is to start looking at software risk analysis on a component-by-component, tier-by-tier, and environment-by-environment level and then apply the principles of measuring threats, vulnerabilities, and impacts at each level.

A practical application risk-analysis approach

At the design stage, any risk-analysis process should be tailored to software design. Recall that the object of this exercise is to determine specific vulnerabilities and threats that exist for the software and assess their impact. A functional decomposition of the application into major components, processes, data stores, and data communication flows, mapped against the environments across which the software will be deployed, allows for a desktop review of threats and potential vulnerabilities. We cannot overemphasize the importance of

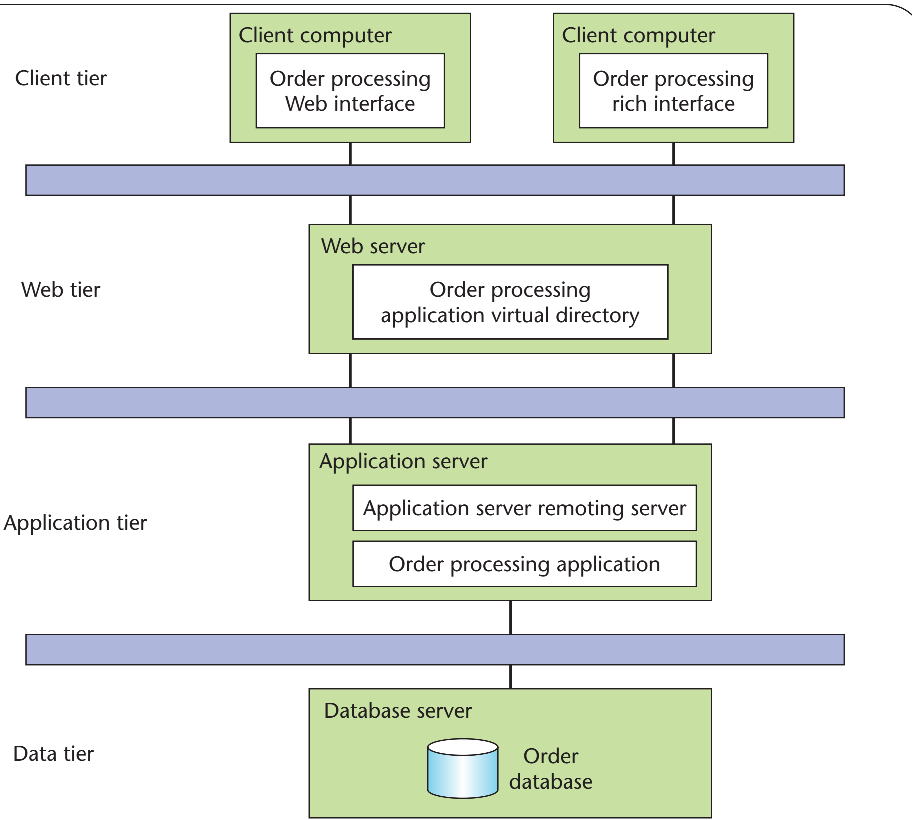

Figure 2. Forest-level view of a standard-issue four-tier Web application. In this design, the client tier exists out on the Internet, while the remaining tiers are on internal networks connected to the Internet. Customers using the client indirectly affect data in the database, so control and access must be managed through all tiers.

using a forest-level view of a system during risk analysis. Some sort of high-level model of the system (from a whiteboard with boxes and arrows to a formally specified mathematical model) makes risk analysis at the architectural level possible.

Although we could contemplate using modeling languages such as UMLSec to attempt to model threats, even the most rudimentary analysis approaches can yield meaningful results. Consider Figure 2, which shows a simple four-tier deployment design pattern for a standard-issue Web-based application. If we apply risk-analysis principles to this level of design, we can immediately draw some useful conclusions about the application's security design.

During the risk-analysis process, we use the high-level design to consider

- the threat present in each tier's environment;
- the kinds of vulnerabilities that might exist in each component as well as the dataflows;
- the business impact of such technical risks, were they to be realized;
- the probability of such a risk being realized; and
- any feasible countermeasures that could be implemented at each tier, taking into account the full range of protection mechanisms available (from base operating system-level security through virtual machine security mechanisms such as the use of Java cryptography extensions in J2EE).

In the simple example shown in Figure 2, each tier exists in a different security realm or trust zone. This fact immediately gives us the context of the threat each tier faces. If we go on to superimpose data types (such as user-logon credentials, records, and orders), their flows (logon requests, record queries, and order entries), and, more importantly, their security classifications, we can draw conclusions about the protection for these data elements and their transmission given the current design.

Suppose that SSL protects user-logon flows between the client and the Web server. Our deployment pattern indicates that although the encrypted tunnel terminates at this tier (because of the inherent threat in the zones occupied by the Web and application tiers), we really must prevent eavesdropping inside and between these two tiers as well. This might indicate the need to establish yet another encrypted tunnel or to consider a different approach to securing this data (maybe message-level encryption instead of tunneling).

Considering the communications risks, it becomes clear why a deployment pattern is valuable, because it lets us consider infrastructure (operating system and network) security mechanisms and application-level mechanisms as risk-mitigation measures.

Decomposing software on a component-by-component basis to establish trust zones is a comfortable way for most software developers and auditors to begin adopting a risk-management approach to software security. Because most systems, especially those exhibiting the *n*-tier architecture, rely on several third-party components and a variety of programming languages, defining zones of trust and taking an outside/in perspective similar to the one normally found in traditional security has clear benefits. In any case, interaction of different products and languages is an architectural element likely to be a vulnerability hotbed.

At its heart, decomposition is a natural way to partition a system. Given a simple decomposition, security professionals will be able to advise developers and architects

about aspects of security they're familiar with, such as network-based component boundaries and authentication. However, the composition problem is unsolved and very tricky—even the most secure components can be assembled into an insecure mess.

As organizations become adept at identifying vulnerability and its business impact, the risk-analysis team should evolve the basic approach to include additional assessment of the risks found within—or encompassing all—tiers. This evolution can uncover technology-specific vulnerabilities based on failings other than trust issues across tier boundaries. Examples of more subtle risks that can only be flushed out with a more sophisticated approach include transaction management risks and luring attacks.

Risk analysis is, at best, a good general-purpose yardstick by which we can judge our security design's effectiveness. Because roughly 50 percent of security problems are the result of design flaws, performing a risk analysis at the design level is an important part of a solid software security program. Taking the trouble to apply risk-analysis methods at the design level for any application often yields valuable, business-relevant results. The process of risk analysis is continuous and applies to many different levels, at once identifying system-level vulnerabilities, assigning probability and impact, and determining reasonable mitigation strategies. By considering the resulting ranked risks, business stakeholders can determine how to manage particular risks and what the most cost-effective controls might be. □

Acknowledgments

We thank John Steven and Stan Wisseman (both of Cigital) for their insightful comments on early drafts of this work. We also thank Bruce Phillips of Fidelity National Financial.

References

1. G. McGraw, "Software Security," *IEEE Security & Privacy*, vol. 2, no. 2, 2004, pp. 80–83.
2. H. Cavusoglu, B. Mishra, and S. Raghunathan, *The Effect of Internet Security Breach Announcements on Market Value of Breached Firms and Internet Security Developers*, tech. report, Univ. of Texas at Dallas, School of Management, Feb. 2002; www.utdallas.edu/~huseyin/breach.pdf.
3. M. Howard and D. LaBlanc, *Writing Secure Code*, 2nd ed., Microsoft Press, 2003.
4. G. Hoglund and G. McGraw, *Exploiting Software*, Addison-Wesley, 2004.
5. J. Viega and G. McGraw, *Building Secure Software: How to Avoid Security Problems the Right Way*, Addison-Wesley, 2001.
6. G. Sindre and A.L. Opdahl, "Eliciting Security Requirements by Misuse Cases," *Proc. 37th Technology of Object-Oriented Languages and Systems* (TOOLS-37), IEEE CS Press, 2000.

***Denis Verdon** is senior vice president of corporate information security at Fidelity National Financial. He has 21 years experience in Information Security and IT, much of it gained while working both as a senior information security executive and as a consultant to senior security executives at Global 200 companies across 19 countries. Contact him at denis.verdon@fnf.com.*

***Gary McGraw** is chief technology officer of Cigital. His real-world experience is grounded in years of consulting with major corporations and software producers. He serves on the technical advisory boards of Authentica, Counterpane, Fortify, and Indigo. He also is coauthor of* Exploiting Software *(Addison-Wesley, 2004),* Building Secure Software *(Addison-Wesley, 2001),* Java Security *(John Wiley & Sons, 1996), and four other books. He has a BA in philosophy from the University of Virginia and a dual PhD in computer science and cognitive science from Indiana University. Contact him at gem@cigital.com.*