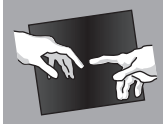
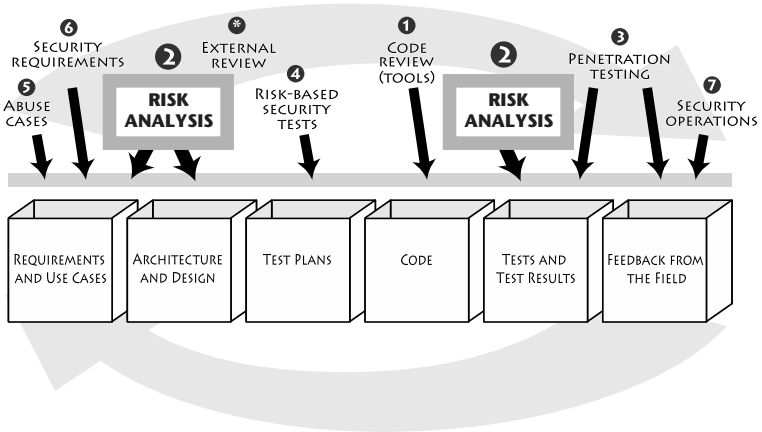


5 Architectural Risk Analysis[1]



Architecture is the learned game, correct and magnificent, of forms assembled in the light.

LE CORBUSIER

Design flaws account for 50% of security problems. You can't find design defects by staring at code—a higher-level understanding is required. That's why architectural risk analysis plays an essential role in any solid software security program. By explicitly identifying risk, you can create a good general-purpose measure of software security, especially if you track risk over time. Because quantifying impact is a critical step in any risk-based approach, risk analysis is a natural way to tie technology issues and concerns directly to the business. A superior risk analysis explicitly links system-level concerns to probability and impact measures that matter to the organization building the software.

[1]Parts of this chapter appeared in original form in *IEEE Security & Privacy* magazine co-authored with Denis Verdon [Verdon and McGraw 2004].

The security community is unanimous in proclaiming the importance of a risk-based approach to security. "Security is risk management" is a mantra oft repeated and yet strangely not well understood. Nomenclature remains a persistent problem in the security community. The term *risk management* is applied to everything from threat modeling and architectural risk analysis to large-scale activities tied up in processes such as RMF (see Chapter 2).

As I describe in Chapter 1, a continuous risk management process is a necessity. This chapter is not about continuous risk management, but it does assume that a base process like the RMF exists and is in place.[2] By teasing apart architectural risk analysis (the critical software security best practice described here) and an overall RMF, we can begin to make better sense of software security risk.

Common Themes among Security Risk Analysis Approaches

Risk management has two distinct flavors in software security. I use the term *risk analysis* to refer to the activity of identifying and ranking risks at some particular stage in the software development lifecycle. Risk analysis is particularly popular when applied to architecture and design-level artifacts. I use the term *risk management* to describe the activity of performing a number of discrete risk analysis exercises, tracking risks throughout development, and strategically mitigating risks. Chapter 2 is about the latter.

A majority of risk analysis process descriptions emphasize that risk identification, ranking, and mitigation is a continuous process and not simply a single step to be completed at one stage of the development lifecycle. Risk analysis results and risk categories thus drive both into requirements (early in the lifecycle) and into testing (where risk results can be used to define and plan particular tests).

Risk analysis, being a specialized subject, is not always best performed solely by the design team without assistance from risk professionals outside the team. Rigorous risk analysis relies heavily on an understanding of business impact, which may require an understanding of laws and regulations as much as the business model supported by the software. Also, human nature dictates that developers and designers will have built up certain assumptions regarding their system and the risks that it faces. Risk and security specialists can at a minimum assist in challenging those assumptions against

[2] All of the other touchpoint chapters make this same assumption.

generally accepted best practices and are in a better position to "assume nothing." (For more on this, see Chapter 9.)

A prototypical risk analysis approach involves several major activities that often include a number of basic substeps.

- **Learn as much as possible about the target of analysis.**
  - Read and understand the specifications, architecture documents, and other design materials.
  - Discuss and brainstorm about the target with a group.
  - Determine system boundary and data sensitivity/criticality.
  - Play with the software (if it exists in executable form).
  - Study the code and other software artifacts (including the use of code analysis tools).
  - Identify threats and agree on relevant sources of attack (e.g., will insiders be considered?).
- **Discuss security issues surrounding the software.**
  - Argue about how the product works and determine areas of disagreement or ambiguity.
  - Identify possible vulnerabilities, sometimes making use of tools or lists of common vulnerabilities.
  - Map out exploits and begin to discuss possible fixes.
  - Gain understanding of current and planned security controls.[3]
- **Determine probability of compromise.**
  - Map out attack scenarios for exploits of vulnerabilities.
  - Balance controls against threat capacity to determine likelihood.
- **Perform impact analysis.**
  - Determine impacts on assets and business goals.
  - Consider impacts on the security posture.
- **Rank risks.**
- **Develop a mitigation strategy.**
  - Recommend countermeasures to mitigate risks.
- **Report findings.**
  - Carefully describe the major and minor risks, with attention to impacts.
  - Provide basic information regarding where to spend limited mitigation resources.

[3]Note that security controls can engender and introduce new security risks themselves (through bugs and flaws) even as they mitigate others.

A number of diverse approaches to risk analysis for security have been devised and practiced over the years. Though many of these approaches were expressly invented for use in the network security space, they still offer valuable risk analysis lessons. The box Risk Analysis in Practice lists a number of historical risk analysis approaches that are worth considering.

My approach to architectural risk analysis fits nicely with the RMF described in Chapter 2. For purposes of completeness, a reintroduction to the RMF is included in the box Risk Analysis Fits in the RMF.

Risk Analysis in Practice

A number of methods calculate a nominal value for an information asset 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.

Examples of risk analysis methodologies for software fall into two basic categories: commercial and standards-based.

Commercial

- STRIDE from Microsoft <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vbcon/html/vbconOverviewOfWebApplicationSecurityThreats.asp> (also see [Howard and LeBlanc 2003])
- Security Risk Management Guide, also from Microsoft <http://www.microsoft.com/technet/security/topics/policiesandprocedures/secrisk/default.mspx>
- ACSM/SAR (Adaptive Countermeasure Selection Mechanism/Security Adequacy Review) from Sun (see [Graff and van Wyk 2003] for public discussion)
- Cigital's architectural risk analysis process (described later in this chapter), which is designed to fit into the RMF (see Chapter 2)

Standards-Based

- ASSET (Automated Security Self-Evaluation Tool) from the National Institute on Standards and Technology (NIST) <http://csrc.nist.gov/asset/>
- OCTAVE (Operationally Critical Threat, Asset, and Vulnerability Evaluation) from SEI <http://www.sei.cmu.edu/publications/documents/99.reports/99tr017/99tr017abstract.html>
- COBIT (Control Objectives for Information and Related Technology) from Information Systems Audit and Control Association (ISACA) <http://www.isaca.org/Template.cfm?Section=COBIT_Online&Template=/ContentManagement/ContentDisplay.cfm&ContentID=15633>

Risk Analysis Fits in the RMF

Architectural risk analysis fits within a continuous risk management framework (RMF) just as the other touchpoint best practices do. The continuous risk management process we use at Cigital loops constantly and at many levels of description through several stages (Figure 5–1). A simplified version of the RMF shown here is described in gory detail in Chapter 2. In this approach, business goals determine risks, risks drive methods, methods yield measurement, measurement drives decision support, and decision support drives fix/rework and application quality.

During the process of architectural risk analysis, we follow basic steps very similar to those making up the RMF.

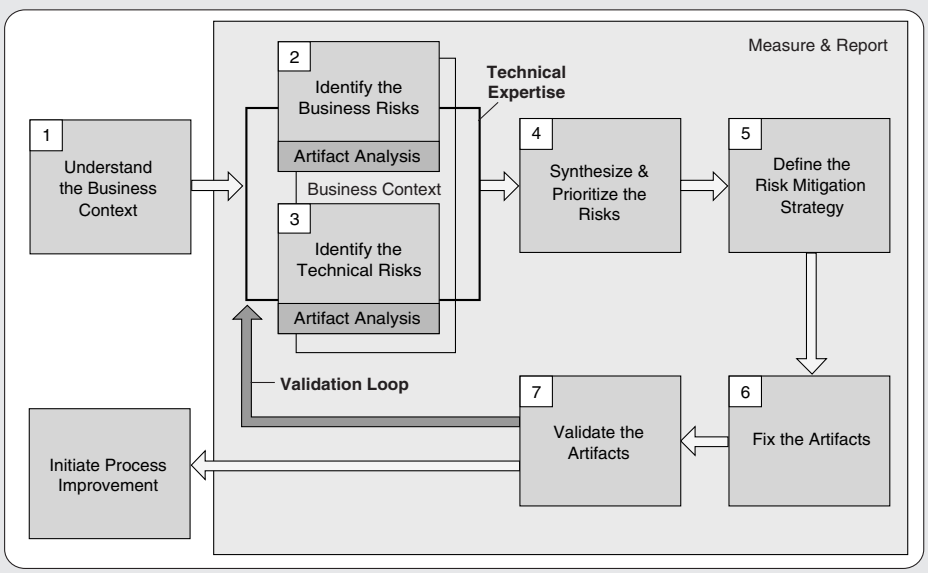


Figure 5–1 Cigital's risk management framework typifies the fractal and continuous nature of risk analysis processes. Many aspects of frameworks like these can be automated—for example, risk storage, business risk to technical risk mapping, and display of status over time.

The RMF shown in Figure 5–1 has a clear loop, called the *validation loop*. This loop is meant to graphically represent the idea that risk management is a continuous process. That is, identifying risks only once in a project is insufficient. The idea of "crossing off a particular stage" once it has been executed and never doing those activities

Continued

again is incorrect. Though the seven stages are shown in a particular serial order in Figure 5–1, they may need to be applied over and over again throughout a software development effort, and their particular ordering may be interleaved in many different ways.

Risk management is in some sense fractal. In other words, the entire continuous, ongoing process can be applied at several different levels. The primary level is the project level. Each stage of the validation loop clearly must have some representation during a complete development effort in order for risk management to be effective. Another level is the software lifecycle artifact level. The validation loop will most likely have a representation given requirements, design, architecture, test plans, and so on. The validation loop will have a representation during both requirements analysis and use case analysis, for example. Fortunately, a generic description of the validation loop as a serial looping process is sufficient to capture critical aspects at all of these levels at once. (See Chapter 2.)

Traditional Risk Analysis Terminology

An in-depth analysis of all existing risk analysis approaches is beyond the scope of this book; instead, I summarize 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 the following:

- Financial loss methodologies that seek to provide a loss figure to be balanced against the cost of implementing various controls
- Mathematically derived "risk ratings" that equate risk to arbitrary ratings for threat, probability, and impact
- Qualitative assessment techniques that base risk assessment on anecdotal or knowledge-driven factors

Each basic approach has its merits, but even when approaches differ in the details, almost all of them share some common concepts that are valuable and should be considered in *any* risk analysis. These commonalities can be captured in a set of basic definitions.

- **Asset**: The object of protection efforts. This may be variously defined as a system component, data, or even a complete system.
- **Risk**: The probability that an asset will suffer an event of a given

negative impact. Various factors determine this calculation: the ease of executing an attack, the motivation and resources of an attacker, the existence of vulnerabilities in a system, and the cost or impact in a particular business context. Risk = probability × impact.

- **Threat**: The actor or agent who is the source of danger. Within information security, this is invariably the danger posed by a malicious agent (e.g., fraudster, attacker, malicious hacker) for a variety of motivations (e.g., financial gain, prestige). Threats carry out attacks on the security of the system (e.g., SQL injection, TCP/IP SYN attacks, buffer overflows, denial of service). Unfortunately, Microsoft has been misusing the term *threat* as a substitute for *risk*. This has led to some confusion in the commercial security space. (See the next box, On Threat Modeling versus Risk Analysis: Microsoft Redefines Terms.)
- **Vulnerability**: For a threat to be effective, it must act against a vulnerability in the system. In general, a vulnerability is a defect or weakness in system security procedures, design, implementation, or internal controls that can be exercised and result in a security breach or a violation of security policy. A vulnerability may exist in one or more of the components making up a system. (Note that the components in question are not necessarily involved with security functionality.) Vulnerability data for a given software system are most often compiled from a combination of OS-level and application-level vulnerability test results (often automated by a "scanner," such as Nessus, Nikto, or Sanctum's Appscan), code reviews, and higher-level architectural reviews. In software, vulnerabilities stem from defects and come in two basic flavors: *flaws* are design-level problems leading to security risk, and *bugs* are implementation-level problems leading to security risk. Automated source code analysis tools tend to focus on bugs. Human expertise is required to uncover flaws.
- **Countermeasures or safeguards**: The management, operational, and technical controls prescribed for an information system which, taken together, adequately protect the confidentiality, integrity, and availability of the system and its information. For every risk, controls may be put in place that either prevent or (at a minimum) detect the risk when it triggers.
- **Impact**: The impact on the organization using the software, were the risk to be realized. This can be monetary or tied to reputation, or may result from the breach of a law, regulation, or contract. Without a quantification of impact, technical vulnerability is hard to deal

with—especially when it comes to mitigation activities. (See the discussion of the "techno-gibberish problem" in Chapter 2.)

- **Probability**: The likelihood that a given event will be triggered. This quantity is often expressed as a percentile, though in most cases calculation of probability is extremely rough. I like to use three simple buckets: high (H), medium (M), and low (L). Geeks have an unnatural propensity to use numbers even when they're not all that useful. Watch out for that when it comes to probability and risk. Some organizations have five, seven, or even ten risk categories (instead of three). Others use exact thresholds (70%) and pretend-precision numbers, such as 68.5%, and end up arguing about decimals. Simple categories and buckets seem to work best, and they emerge from the soup of risks almost automatically anyway.

Using these basic definitions, risk analysis approaches diverge on how to arrive at particular values for these attributes. A number of methods calculate a nominal value for an information asset and attempt to determine risk as a function of loss and event probability. Some methods use checklists of risk categories, threats, and attacks to ascertain risk.

On Threat Modeling versus Risk Analysis: Microsoft Redefines Terms

The good news is that Microsoft appears to be taking software security very seriously. The company has its own set of experts (the superstar being Michael Howard) and has even invented its own processes (paramount among these being the STRIDE model). The bad news is that the company also has its own vocabulary, which differs in important ways from standard usage in the security literature.

The biggest problem lies in misuse of the term *threat*. Microsoft describes as *threat modeling* what most others call *risk analysis*. For example, in the book *Threat Modeling,* Swiderski and Snyder explain that:

> *During threat modeling, the application is dissected into its functional components. The development team analyzes the components at every entry point and traces data flow through all functionality to identify security weaknesses.* [Swiderski and Snyder 2004, p. 16]

Clearly they are describing risk analysis. The term *threat modeling* should really refer to the activity of describing and cataloging threats—those actors or agents who want to

attack your system. Having an old-style threat model like this is a critical step in thinking about security risk. After all, all the security vulnerabilities and software defects in the world would not matter if nobody were hell-bent on exploiting them.

The Microsoft Approach

Big problems with vocabulary aside, the basic process described in the book *Threat Modeling* is sound and well worth considering. Based on the STRIDE model introduced by Howard and LeBlanc (also from Microsoft), the Microsoft risk analysis process relies a bit too heavily on the notion of cycling through a list of attacks [Howard and LeBlanc 2003]. For example, STRIDE is an acronym for Spoofing, Tampering, Repudiation, Information disclosure, Denial of service, and Elevation of privilege. These are categories of attacks, and like attack patterns, they make useful lists of things to consider when identifying risks. Of course, any list of attacks will be incomplete and is very much unlikely to cover new creative attacks.* In any case, applying the STRIDE model in practice is an exercise in "sliding" known attacks over an existing design and seeing what matches. This is an excellent thing to do.

Risk analysis is the act of creating security-relevant design specifications and later testing that design. This makes it an integral part of building any secure system. The *Threat Modeling* book describes how to build a model of the system using both data flow diagrams and use cases. Then it goes on to describe a simple process for creating attack hypotheses using both lists of vulnerabilities and lists of system assets as starting points. This process results in attack trees similar in nature to the attack trees described in *Building Secure Software* [Viega and McGraw 2001].

Go ahead and make use of Microsoft's process, but please don't call it threat modeling.

*You can think of these checklists of attacks as analogous to virus patterns in a virus checker. Virus checkers are darn good at catching known viruses and stopping them cold. But when a new virus comes out and is not in the "definition list," watch out!

Knowledge Requirement

Architectural risk analysis is knowledge intensive. For example, Microsoft's STRIDE model involves the understanding and application of several risk categories during analysis[4] [Howard and LeBlanc 2003]. Similarly, my risk

[4]In STRIDE, these are referred to as "threat categories"; however, that term would more correctly be used to refer to groups of attackers, not to groups of risks.

analysis approach involves three basic steps (described more fully later in the chapter):

1. Attack resistance analysis
2. Ambiguity analysis
3. Weakness analysis

Knowledge is most useful in each of these steps: the use of attack patterns [Hoglund and McGraw 2004] and exploit graphs for understanding *attack resistance analysis,* knowledge of design principles for use in *ambiguity analysis* [Viega and McGraw 2001], and knowledge regarding security issues in commonly used frameworks (.NET and J2EE being two examples) and other third-party components to perform *weakness analysis*. These three subprocesses of my approach to risk analysis are discussed in detail in this chapter.

For more on the kinds of knowledge useful to all aspects of software security, including architectural risk analysis, see Chapter 11.

The Necessity of a Forest-Level View

A central activity in design-level risk analysis involves building up a consistent view of the target system at a reasonably high level. The idea is to see the forest and not get lost in the trees. The most appropriate level for this description is the typical whiteboard view of boxes and arrows describing the interaction of various critical components in a design. For one example, see the following box, .NET Security Model Overview.

Commonly, not enough of the many people often involved in a software project can answer the basic question, "What does the software do?" All too often, software people play happily in the weeds, hacking away at various and sundry functions while ignoring the big picture. Maybe, if you're lucky, one person knows how all the moving parts work; or maybe nobody knows. A one-page overview, or "forest-level" view, makes it much easier for everyone involved in the project to understand what's going on.

The actual form that this high-level description takes is unimportant. What is important is that an analyst can comprehend the big picture and use it as a jumping-off place for analysis. Some organizations like to use UML (the Unified Modeling Language) to describe their systems.[5] I believe UML

[5]For more on UML, see <http://www.uml.org/>.

is not very useful, mostly because I have seen it too often abused by the high priests of software obfuscation to hide their lack of clue. But UML may be useful for some. Other organizations might like a boxes-and-arrows picture of the sort described here. Formalists might insist on a formal model that can be passed into a theorem prover in a mathematical language like Z. Still others might resort to complex message-passing descriptions—a kind of model that is particularly useful in describing complex cryptosystems. In the end, the particular approach taken must result in a comprehensible high-level overview of the system that is as concise as possible.

The nature of software systems leads many developers and analysts to assume (incorrectly) that code-level description of software is sufficient for spotting design problems. Though this may 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. Because the XP guys all started out as Smalltalk programmers they may be a bit confused about whether the code is the design. A quick look at the results of the obfuscated C contest <http://www.ioccc.org> should disavow them of this belief.[6]

Without a whiteboard level of description, an architectural risk analysis is likely to overlook important risks related to flaws. Build a forest-level overview as the first thing you do in any architectural risk analysis.

.NET Security Model Overview

Figure 5–2 shows a one-page high-level architectural view of the .NET security model prepared while performing a .NET risk analysis. Before this diagram was created, the only high-level description of the .NET security architecture was a book-length description of its (way too many) parts. Putting all the parts together in one picture is an essential aspect of risk analysis.

All risk analyses should begin by understanding and, if necessary, describing and documenting a high-level overview of the system to be analyzed. Sometimes the act of building this picture is a monumental undertaking. Sometimes a one-page overview already exists. In any case, making one is a great idea.

Continued

[6]Incidentally, any language whose aficionados purposefully revel in its ability to be incomprehensible (even to the initiated) has serious issues. Perhaps experienced developers should require a license to use C. Newbies would not be permitted until properly licensed.

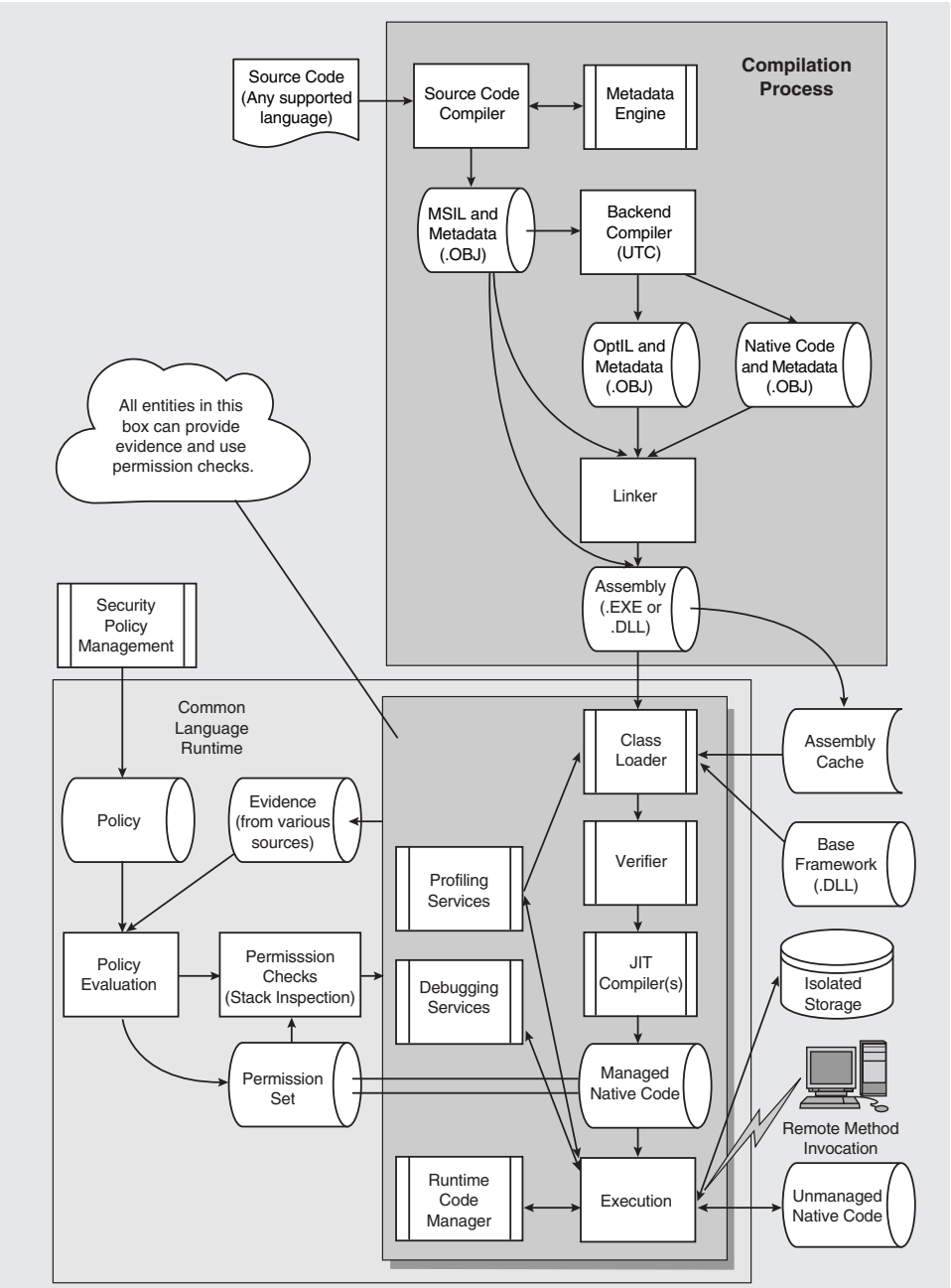


Figure 5–2 A one-page overview of Microsoft's .NET security model. An architectural picture like this, though not in any sense detailed enough to perform a complete analysis, is extremely useful for thinking about components, modules, and possible attacks. Every one-page overview should list all components and show what is connected to what.

By referencing the picture in Figure 5–2, an analyst can hypothesize about possible attacks. This can be driven by a list of known attacks such as the attack patterns described in Chapter 8 (and fleshed out in vivid detail in *Exploiting Software* [Hoglund and McGraw 2004]), or it can be driven by deep technical understanding of the moving parts.

As an example of the latter approach, consider the flow of information in Figure 5–2. In this picture the *Verifier* feeds the just in time (*JIT) compiler*. As noted in *Java Security,* the Verifier exists to ensure that the bytecode (in this case, CLR code) coheres to various critical type-safety constraints [McGraw and Felten 1996]. Type safety is about objects having certain properties that can be guaranteed. If type-safety rules are not followed or the Virtual Machine becomes confused about type safety, very bad things happen.

Anyway, the Verifier does its thing and passes information on to the JIT compiler.

A JIT compiler transforms intermediate CLR code (or Java bytecode) into native code (usually x86 code) "just in time." This is done for reasons of speed. For the security model to retain its potency, the JIT compiler must carry out only transformations that preserve type safety. By thinking through scenarios in which the JIT compiler breaks type safety, we can anticipate attacks and identify future risks. Interestingly, several relevant security issues based on this line of reasoning about attacks and type safety led to the discovery of serious security problems in Java. (For a complete description of the Java attacks, see <http://www.securingjava.com>, where you can find a complete, free, online edition of my book *Securing Java* [McGraw and Felten 1999].)

Unless we built up a sufficient high-level understanding of the .NET security model (probably through the process of creating our one-page picture), we would not likely come across possible attacks like the one described here.

One funny story about forest-level views is worth mentioning. I was once asked to do a security review of an online day-trading application that was extremely complex. The system involved live online attachments to the ATM network and to the stock exchange. Security was pretty important. We had trouble estimating the amount of work to be involved since there was no design specification to go on.[7] We flew down to Texas and got started anyway. Turns out that only one person in the entire hundred-person company knew how the system actually worked and what all the moving parts

[7]The dirty little trick of software development is that without a design spec your system can't be wrong, it can only be surprising! Don't let the lack of a spec go by without raising a ruckus. Get a spec.

were. The biggest risk was obvious! If that one person were hit by a bus, the entire enterprise would grind to a spectacular halt. We spent most of the first week of the work interviewing the architect and creating both a forest-level view and more detailed documentation.

A Traditional Example of a Risk Calculation

One classic method of risk analysis expresses risk as a financial loss, or Annualized Loss Expectancy (ALE), 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 predicted frequency of a loss event happening).

Consider an Internet-based equities trading application possessing a vulnerability that may result in unauthorized access, with the implication that unauthorized stock trades can be made. Assume that 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 the trade. We'll assign a cost of $150 for any such event. This yields an SLE = $150. With even an ARO of 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, in the case of 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 would dwarf such an annual loss figure).[8]

Other methods take a more qualitative route. In the case of a Web server providing a company's face to the world, a Web site 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 [Cavusoglu, Mishra, and Raghunathan 2002]). In cases where intangible assets are involved (e.g., reputation), qualitative risk assessment may be a more appropriate way to capture loss.

[8]There are other quantitative methods that don't use ALE. For example, some organizations use hard numbers such as the actual cost of developing and operating the system, dollar value to paying customers, and so on.

Regardless of the technique used, most practitioners advocate a return-on-investment study to determine whether a given countermeasure is a cost-effective method for achieving the desired security goal. For example, adding applied cryptography to an application server, using native APIs (e.g., MS-CAPI) without the aid of dedicated hardware acceleration, may be cheap in the short term; but if this results in a significant loss in transaction volume throughput, a better ROI may be achieved by investing up front in crypto acceleration hardware. (Make sure to be realistic about just what ROI means if you choose to use the term. See the box The Truth about ROI.)

Interested organizations are advised to adopt the risk calculation methodology that best reflects their needs. The techniques described in this chapter provide a starting point.

The Truth about ROI

ROI sounds great in glossy marketing handouts. But what exactly does ROI mean for security? Other than confirming that getting started with security early in the lifecycle is of critical importance and will save you money, studies of return on security investment (ROSI) have not amounted to much.

Fact is, security is more like insurance than it is like some kind of investment. You can manage risk by identifying and mitigating security issues both technically and at the business level. But you will never hit a "big payoff" if your security holds. You'll only avoid serious negative consequences if it doesn't. We buy car insurance for just that reason: not because we can't wait for the big payoff when we have a crash but just in case we do.

Limitations of Traditional Approaches

Traditional risk analysis output is difficult to apply directly to modern software design. For example, in the quantitative risk analysis equation described in the previous section, even assuming a high level of confidence in the ability to predict the dollar loss for a given event and having performed Monte Carlo distribution analysis of prior events to derive a statistically sound probability distribution for future events, there's still a large gap between the raw dollar figure of an ALE and a detailed software security mitigation definition.

Another, 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 be concerned about at a component/environment level. This is where a large knowledge base and lots of experience is invaluable. (See Chapter 11 for more on software security knowledge.)

The thorny knowledge problem arises in part because modern applications, including Web Services applications, are designed to span multiple boundaries of trust. Vulnerability of, and risk to, any given component varies with the platform that the component exists on (e.g., C# applications on Windows .NET Server versus J2EE applications on Tomcat/Apache/Linux) and with the environment it exists in (secure production network versus client network versus Internet DMZ). However, few of the traditional approaches adequately address the contextual variability of risk given changes in the core environment. This becomes a fatal flaw when considering highly distributed applications, Service Oriented Architectures, or Web Services.

In modern frameworks, such as .NET and J2EE, security methods exist at almost every layer of the OSI model, yet too many applications today rely on a "reactive protection" infrastructure (e.g., firewalls, SSL) that provides protection below layer four only. This is too often summed up in the claim "We are secure because we use SSL and implement firewalls," leaving open all sorts of questions such as those engendered by port 80 attacks, SQL injection, class spoofing, and method overwriting (to name a handful).

One answer to this problem is to begin to look at software risk analysis on a component-by-component, tier-by-tier, environment-by-environment level and apply the principles of measuring threats, risks, vulnerabilities, and impacts at all of these levels.

Modern Risk Analysis

Given the limitations of traditional approaches, a more holistic risk management methodology involves thinking about risk throughout the lifecycle (as described in Chapter 2). Starting the risk analysis process early is critical. In fact, risk analysis is even effective at the requirements level. Modern approaches emphasize the importance of an architectural view and of architectural risk analysis.

Security Requirements

In the purest sense, risk analysis starts at the requirements stage because design requirements should take into account the risks that you are trying to counter. The box Back to Requirements briefly covers three approaches to interjecting a risk-based philosophy into the requirements phase. (Do note that the requirements systems based around UML tend to focus more attention on security *functionality* than they do on abuse cases, which I discuss at length in Chapter 8.)

Whatever risk analysis method is adopted, the requirements process should be driven by risk.

Back to Requirements

SecureUML* is a methodology for modeling access control policies and their integration into a model-driven software development process. SecureUML is based on Role-Based Access Control and models security requirements for well-behaved applications in predictable environments.

UMLsec [Jurjens 2001] is an extension of UML to include modeling of security-related features, such as confidentiality and access control.

Sindre and Opdahl [2000] attempt to model abuse cases as a way of understanding how an application might respond to threats in a less controllable environment and to describe functions that the system should not allow.

*See http://kisogawa.inf.ethz.ch/WebBIB/publications-softech/papers/2002/0_secuml_uml2002.pdf.

As stated earlier, a key variable in the risk equation is *impact*. The business impacts of any risks that we are trying to avoid can be many, but for the most part, they boil down into three broad categories:

1. Legal and/or regulatory risk: These may include federal or state laws and regulations (e.g., the Gramm-Leach-Bliley Act [GLBA], HIPPA, or the now-famous California Senate Bill 1386, also known as SB1386)
2. Financial or commercial considerations (e.g., protection of revenue, control over high-value intellectual property, preservation of brand and reputation)

3. Contractual considerations (e.g., service-level agreements, avoidance of liability)

Even at this early point in the lifecycle, the first risk-based decisions should be made. One approach might be to break down requirements into three simple categories: "must-haves," "important-to-haves," and "nice-but-unnecessary-to-haves."

Unless you are running an illegal operation, laws and regulations should always be classed into the first category, making these requirements instantly mandatory and not subject to further risk analysis (although an ROI study should always be conducted to select the most cost-effective mitigations). For example, if the law requires you to protect private information, this is mandatory and should not be the subject of a risk-based decision. Why? Because the government may have the power to put you out of business, which is the mother of all risks (and if you want to test the government and regulators on this one, then go ahead—just don't say that you weren't warned!).

You are then left with risk impacts that need to be managed in other ways, the ones that have as variables potential impact and probability. At the initial requirements definition stage, you may be able to make some assumptions regarding the controls that are necessary and the ones that may not be.

Even application of these simple ideas will put you ahead of the majority of software developers. Then as we move toward the design and build stages, risk analysis should begin to test those assumptions made at the requirements stage by analyzing the risks and vulnerabilities inherent in the design. Finally, tests and test planning should be driven by risk analysis results as well.

A Basic Risk Analysis Approach

To encompass the design stage, any risk analysis process should be tailored. The object of this tailoring exercise is to determine specific vulnerabilities and risks that exist for the software. 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. I cannot overemphasize the importance of using a forest-level view of a system during risk analysis. Some sort of high-level model of the system (from a whiteboard boxes-and-arrows picture to a formally

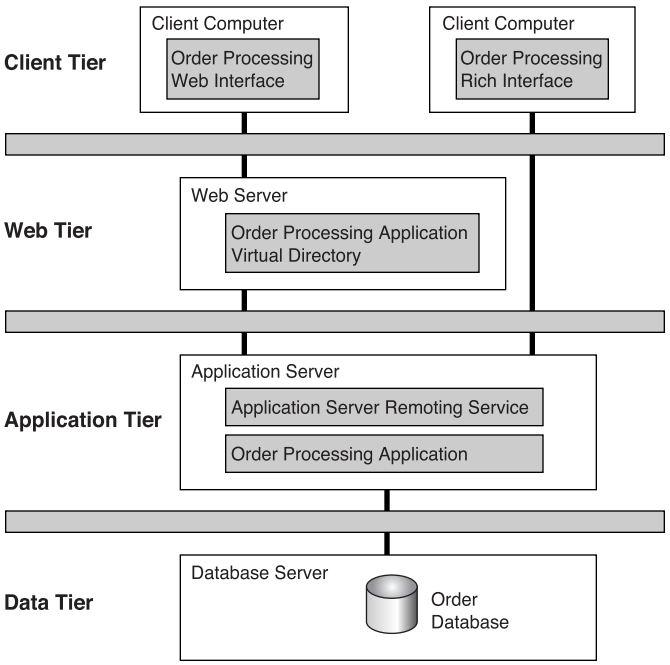


Figure 5–3 A forest-level view of a standard-issue four-tier Web application.

specified mathematical model) makes risk analysis at the architectural level possible.

Although one could contemplate using modeling languages, such as UMLsec, to attempt to model risks, even the most rudimentary analysis approaches can yield meaningful results. Consider Figure 5–3, 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 security design of the application.

During the risk analysis process we should consider the following:

- The threats who are likely to want to attack our system
- The risks present in each tier's environment
- The kinds of vulnerabilities that might exist in each component, as well as the data flow
- The business impact of such technical risks, were they to be realized
- The probability of such a risk being realized

- Any feasible countermeasures that could be implemented at each tier, taking into account the full range of protection mechanisms available (e.g., from base operating system–level security through Virtual Machine security mechanisms, such as use of the Java Cryptography Extensions in J2EE)

This very basic process will sound familiar if you read Chapter 2 on the RMF. In that chapter, I describe in great detail a number of critical risk management steps in an iterative model.

In this simple example, each of the tiers exists in a different security realm or trust zone. This fact immediately provides us with the context of risk faced by each tier. If we go on to superimpose data types (e.g., user logon credentials, records, orders) and their flows (logon requests, record queries, order entries) and, more importantly, their security classifications, we can draw conclusions about the protection of these data elements and their transmission given the current design.

For example, suppose that user logon flows are protected by SSL between the client and the Web server. However, our deployment pattern indicates that though the encrypted tunnel terminates at this tier, because of the threat inherent in the zones occupied by the Web and application tiers, we really need to prevent eavesdropping inside and between these two tiers as well. This might indicate the need to establish yet another encrypted tunnel or, possibly, to consider a different approach to securing these data (e.g., message-level encryption as opposed to tunneling).

Use of a deployment pattern in this analysis is valuable because it allows us to consider both infrastructure (i.e., operating systems and network) security mechanisms as well as application-level mechanisms as risk mitigation measures.

Realize that 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 that normally observed 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 that they're familiar with such as network-based component boundaries and authentication (as I highlight in the example). Do not forget, however, that the composition problem (putting the components all back together) is unsolved and very tricky, and that even the most secure components can be assembled into an insecure mess!

As organizations become adept at identifying vulnerability and its business impact consistently using the approach illustrated earlier, the approach should be evolved to include additional assessment of risks found within tiers and encompassing all tiers. This more sophisticated approach uncovers technology-specific vulnerabilities based on failings other than trust issues across tier boundaries. Exploits related to broken transaction management and phishing attacks[9] are examples of some of the more subtle risks one might encounter with an enhanced approach.

Finally, a design-level risk analysis approach can also be augmented with data from code reviews and risk-based testing.

Coder's Corner

Avi Rubin, a professor at Johns Hopkins University, and his graduate students spent much effort performing an architectural risk analysis on Diebold electronic voting machines. Their work is collected here <http://avirubin.com/vote/>.

The abstract of their paper <http://avirubin.com/vote.pdf> on one of their more famous (and controversial) analyses says:

> *With significant U.S. federal funds now available to replace outdated punch-card and mechanical voting systems, municipalities and states throughout the U.S. are adopting paperless electronic voting systems from a number of different vendors. We present a security analysis of the source code to one such machine used in a significant share of the market. Our analysis shows that this voting system is far below even the most minimal security standards applicable in other contexts.* We identify several problems including unauthorized privilege escalation, incorrect use of cryptography, vulnerabilities to network threats, and poor

Continued

[9]For more on phishing, which combines social engineering and technical subterfuge, see <http://www.antiphishing.org/>.

software development processes *[emphasis added]. We show that voters, without any insider privileges, can cast unlimited votes without being detected by any mechanisms within the voting terminal software. Furthermore, we show that even the most serious of our outsider attacks could have been discovered and executed without access to the source code. In the face of such attacks, the usual worries about insider threats are not the only concerns; outsiders can do the damage. That said, we demonstrate that the insider threat is also quite considerable, showing that not only can an insider, such as a poll worker, modify the votes, but that insiders can also violate voter privacy and match votes with the voters who cast them. We conclude that this voting system is unsuitable for use in a general election. Any paperless electronic voting system might suffer similar flaws, despite any "certification" it could have otherwise received. We suggest that the best solutions are voting systems having a "voter-verifiable audit trail," where a computerized voting system might print a paper ballot that can be read and verified by the voter.*

In the paper, the authors present a number of findings. Before presenting the technical information, a concise overview of the system (a forest-level view) is presented. The overview sets the stage for the technical results, many of which focus on the construction of the system and its architecture. Among the technical results is the following finding:

3.2 Casting multiple votes

In the Diebold system, a voter begins the voting process by inserting a smart card into the voting terminal. Upon checking that the card is "active," the voting terminal collects the user's vote and then deactivates the user's card; the deactivation actually occurs by rewriting the card's type, which is stored as an 8-bit value on the card, from `VOTER_CARD (0x01)` *to* `CANCELED_CARD (0x08)`*. Since an adversary can make perfectly valid smart cards, the adversary could bring a stack of active cards to the voting booth. Doing so gives the adversary the ability to vote multiple times. More simply, instead of bringing multiple cards to the voting booth, the adversary could program a smart card to ignore the voting terminal's deactivation command. Such an adversary could use one card to vote multiple times. Note here that the adversary could be a regular voter, and not necessarily an election insider.*

Will the adversary's multiple-votes be detected by the voting system?

To answer this question, we must first consider what information is encoded on the voter cards on a per voter basis. The only per voter information is a "voter serial number" (`m_VoterSN`* in the* `CVoterInfo` *class).* `m_VoterSN` *is only recorded by the voting terminal if the voter decides* not *to place a vote (as noted in the comments in* `TSElection/Results.cpp`*, this field is recorded for uncounted votes for backward compatibility reasons). It is important to note that if a voter decides to cancel his or her vote, the voter will have the opportunity to vote again using that same card (and, after the vote has been cast,* `m_VoterSN` *will no longer be recorded).*

If we assume the number of collected votes becomes greater than the number of people who showed up to vote, and if the polling locations keep accurate counts of the number of people who show up to vote, then the back-end system, if designed properly, should be able to detect the existence of counterfeit votes. However, because `m_VoterSN` *is only stored for those who did not vote, there will be no way for the tabulating system to distinguish the real votes from the counterfeit votes. This would cast serious doubt on the validity of the election results. The solution proposed by one election official, to have everyone vote again, does not seem like a viable solution.*

Notice how the technical result is presented in terms of impact. The key to a good risk analysis is clearly stated impact statements. The only thing missing in the report is a mitigation strategy that is workable. The Diebold people appear to have their software security work cut out for them!

Touchpoint Process: Architectural Risk Analysis

Architectural risk analysis as practiced today is usually performed by experts in an ad hoc fashion. Such an approach does not scale, nor is it in any way repeatable or consistent. Results are deeply constrained by the expertise and experience of the team doing the analysis. Every team does its own thing. For these reasons, the results of disparate analyses are difficult to compare (if they are comparable at all). That's not so good.

As an alternative to the ad hoc approach, Cigital uses the architectural risk analysis process shown in Figure 5–4. This process complements and extends the RMF of Chapter 2. Though the process described here is certainly

not the "be all, end all, one and only" way to carry out architectural risk analysis, the three subprocesses described here are extraordinarily powerful.

A risk analysis should be carried out only once a reasonable, big-picture overview of the system has been established. The idea is to forget about the code-based trees of bugland (temporarily at least) and concentrate on the forest. Thus the first step of the process shown in the figure is to build a one-page overview of the system under analysis. Sometimes a one-page big picture exists, but more often it does not. The one-page overview can be developed through a process of artifact analysis coupled with interviews. Inputs to the process are shown in the leftmost column of Figure 5–4.

Three critical steps (or subprocesses) make up the heart of this architectural risk analysis approach:

1. Attack resistance analysis
2. Ambiguity analysis
3. Weakness analysis

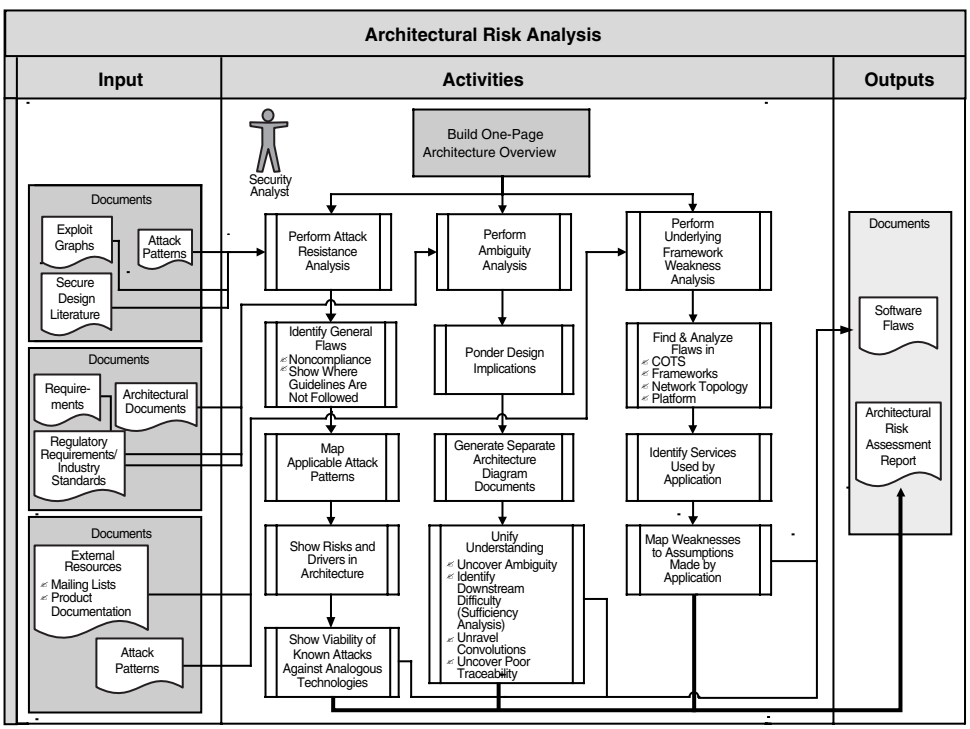


Figure 5–4 A simple process diagram for architectural risk analysis.

Don't forget to refer back to Figure 5–4 as you read about the three subprocesses.

Attack Resistance Analysis

Attack resistance analysis is meant to capture the checklist-like approach to risk analysis taken in Microsoft's STRIDE approach. The gist of the idea is to use information about known attacks, attack patterns, and vulnerabilities during the process of analysis. That is, given the one-page overview, how does the system fare against known attacks? Four steps are involved in this subprocess.

1. Identify general flaws using secure design literature and checklists (e.g., cycling through the Spoofing, Tampering, . . . categories from STRIDE). A knowledge base of historical risks is particularly useful in this activity.
2. Map attack patterns using either the results of abuse case development (see Chapter 8) or a list of attack patterns.
3. Identify risks in the architecture based on the use of checklists.
4. Understand and demonstrate the viability of these known attacks (using something like exploit graphs; see the Exploit Graphs box).

Note that this subprocess is very good at finding *known* problems but is not very good at finding new or otherwise creative attacks.

Example flaws uncovered by the attack resistance subprocess, in my experience, include the following.

- *Transparent authentication token generation/management:* In this flaw, tokens meant to identify a user are easy to guess or otherwise simple to misuse. Web-based programs that use "hidden" variables to preserve user state are a prime example of how not to do this. A number of these flaws are described in detail in *Exploiting Software* [Hoglund and McGraw 2004].
- *Misuse of cryptographic primitives:* This flaw is almost self-explanatory. The best example is the seriously flawed WEP protocol found in 802.11b, which misused cryptography to such an extent that the security was completely compromised [Stubblefield, Ioannides, and Rubin 2004].
- *Easily subverted guard components, broken encapsulation:* Examples here are slightly more subtle, but consider a situation in which an API is subverted and functionality is either misused or used in a surprising new

way. APIs can be thought of as classical "guards" in some cases, as long as they remain a choke point and single point of entry. As soon as they can be avoided, they cease to be useful.

- *Cross-language trust/privilege issues:* Flaws arise when language boundaries are crossed but input filtering and state-preservation mechanisms fail.

Exploit Graphs

An exploit graph helps an analyst understand what kind of access and/or pattern is required to carry out an attack given a software risk. Flowcharts are very useful in describing an exploit and should include some basics such as attack delivery (payloads), gaining access, privilege escalation, subverting protections, descriptions of architectural failure, and discussion of any existing mitigations (and their effectiveness). Charts help. Figure 5–5 shows a simple exploit graph that illustrates a mobile code attack.

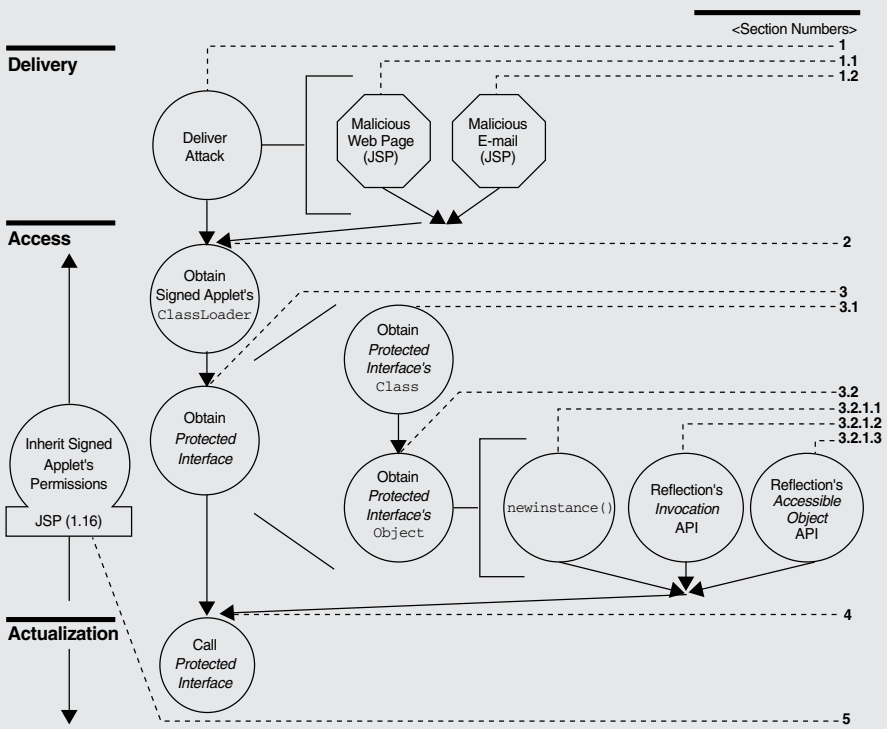


Figure 5–5 An exploit graph showing one of the mobile code attacks described in *Securing Java* [McGraw and Felten 1999]. The section numbers refer to entries in an associated table (in this case, Table 5–1). John Steven of Cigital created this graph.

Table 5–1 A Partial Exploit Graph Table to Accompany Figure 5–5

Step #	Detail: How/What	Conditions	Protection
Delivery 1	Deliver attack: get attack code onto machine with Jewel.	Client must have Internet access.	
Delivery 1.1	Trick user to point browser to JSP.	Browser must have "run JSP" enabled.	Disable JSSP in browser. NOTE**:** doing so prevents other sites from working.
Delivery 1.2	Send victim e-mail containing malicious JSP.	User's mail reader must interpret JSP.	Disable JSP execution in mail reader.

Note: JSP refers to Java Server Page.

Exploit graphs also require some explanation in text as briefly described earlier. Table 5–1 is a partial view (attack delivery only) of the table meant to accompany Figure 5–5.

Though attack graphs are not yet a mechanism in widespread use, they do help in a risk analysis. Their most important contribution lies in allowing an analyst to estimate the level of effort required to exploit a flaw. When it comes to exploit development, having a set of exploit graphs on hand can help determine which one exploit (usually of many) is the best to develop in the case that some kind of "proof" is required. Sometimes you will find that exploit development is required to convince skeptical observers that there is a serious problem that needs to be fixed.

Ambiguity Analysis

Ambiguity analysis is the subprocess capturing the creative activity required to discover *new* risks. This process, by definition, requires at least two analysts (the more the merrier) and some amount of experience. The idea is for each team member to carry out separate analysis activities in parallel. Only after these separate analyses are complete does the team come together in the "unify understanding" step shown in Figure 5–4.

We all know what happens when two or more software architects are put in a room together . . . catfight—often a catfight of world-bending magnitude. The ambiguity analysis subprocess takes advantage of the multiple points of view afforded by the art that is software architecture to create a

critical analysis technique. Where good architects disagree, there lie interesting things (and sometimes new flaws).

In 1998, when performing an architectural risk analysis on early Java Card systems with John Viega and Brad Arkin (their first), my team started with a process very much like STRIDE. The team members each went their solitary analysis ways with their own private list of possible flaws and then came together for a whiteboard brainstorming session. When the team came together, it became apparent that none of the standard-issue attacks considered by the new team members were directly applicable in any obvious fashion. But we could not very well declare the system "secure" and go on to bill the customer (Visa)! What to do?!

As we started to describe together how the system worked (not how it failed, but how it worked), disagreements cropped up. It turns out that these disagreements and misunderstandings were harbingers of security risks. The creative process of describing to others how the system worked (well, at least how we thought it worked) was extremely valuable. Any major points of disagreement or any clear ambiguities became points of further analysis. This evolved into the subprocess of ambiguity analysis.

Ambiguity analysis helps to uncover ambiguity and inconsistency, identify downstream difficulty (through a process of traceability analysis), and unravel convolution. Unfortunately, this subprocess works best when carried out by a team of very experienced analysts. Furthermore, it is best taught in an apprenticeship situation. Perhaps knowledge management collections will make this all a bit less arbitrary (see Chapter 11).

Example flaws uncovered by the ambiguity analysis subprocess in my experience include the following.

- *Protocol, authentication problems:* One example involved key material used to (accidentally) encrypt itself in a complex new crypto system. It turns out that this mistake cut down the possible search space for a key from extremely large to manageably small.[10] This turned out to be a previously unknown attack, but it was fatal.
- *Java Card applet firewall and Java inner class issues:* Two examples. The first was a problematic object-sharing mechanism that suffered from serious transitive trust issues, the gist being that class A shared method foo with class B, and class B could then publish the method to the world (something A did not necessarily condone). The second involved the

[10]That is, breakable in some feasible time period with a standard machine.

way that inner classes were actually implemented (and continue to be implemented) in various Java compilers. Turns out that package scoping in this case was somewhat counterintuitive and that inner classes had a privilege scope that was surprisingly large.

- *Type safety and type confusion:* Type-safety problems in Java accounted for a good portion of the serious Java attacks from the mid-1990s. See *Securing Java* [McGraw and Felten 1999].
- *Password retrieval, fitness, and strength:* Why people continue to roll their own password mechanisms is beyond me. They do, though.

Weakness Analysis

Weakness analysis is a subprocess aimed at understanding the impact of external software dependencies. Software is no longer created in giant monolithic `a.out` globs (as it was in the good old days). Modern software is usually built on top of complex middleware frameworks like .NET and J2EE. Furthermore, almost all code counts on outside libraries like DLLs or common language libraries such as `glibc`. To make matters worse, distributed code—once the interesting architectural exception—has become the norm. With the rapid evolution of software has come a whole host of problems caused by linking in (or otherwise counting on) broken stuff. Leslie Lamport's definition of a distributed system as "one in which the failure of a computer you didn't even know existed can render your own computer unusable" describes exactly why the weakness problem is hard.

Uncovering weaknesses that arise by counting on outside software requires consideration of:

- COTS (including various outside security feature packages like the RSA libraries or Netegrity's authentication modules)
- Frameworks (J2EE, .NET, and any number of other middleware frameworks)
- Network topology (modern software almost always exists in a networked environment)
- Platform (consider what it's like to be application code on a cell phone or a smart card)[11]
- Physical environment (consider storage devices like USB keys and iPods)
- Build environment (what happens when you rely on a broken or poisoned compiler? what if your build machine is running a rootkit?)

[11]Not to mention a smart card living in a cell phone.

In the coming days of Service Oriented Architectures (SOAs), understanding which services your code is counting on and exactly what your code expects those services to deliver is critical. Common components make particularly attractive targets for attack. Common mode failure goes global.

The basic idea here is to understand what kind of assumptions you are making about outside software, and what will happen when those assumptions fail (or are coerced into failing). When assumptions fail, weaknesses are often revealed in stark relief. A large base of experience with third-party software libraries, systems, and platforms is extremely valuable when carrying out weakness analysis. Unfortunately, no perfect clearinghouse of security information for third-party software exists. One good idea is to take advantage of public security discussion forums such as BugTraq <http://www.securityfocus.com/archive/1>, comp.risks <http://catless.ncl.ac.uk/Risks>, and security tracker <http://www.securitytracker.com>.[12]

Example flaws uncovered by the weakness analysis subprocess in my experience include the following.

- *Browser and other VM sandboxing failures*: Browsers are overly complex pieces of software rivaled in complexity only by operating systems. Browsers have so many moving parts that finding unexplored niches and other "between the seams" flaws is easy.
- *Insecure service provision—RMI, COM, and so on*: Protocols and communications systems are often a standard feature of modern software. When Java's RMI was found to fail open <http://www.cs.princeton.edu/~balfanz>, the systems counting on RMI were all subject to the same kind of attack.
- *Debug (or other operational) interfaces*: Debugging code is always as useful to the attacker as it is to the maintainer. Don't send error reports to your (mis)user.
- *Unused (but privileged) product "features"*: If you put overly powerful features into your design, don't be surprised when they are turned against you. See *Building Secure Software* for a good story of what happened when old-fashioned bulletin board systems allowed a user to invoke `emacs` [Viega and McGraw 2001].

[12]Have you ever wondered whether the software you're working on (or counting on) has been successfully attacked? Check out the public mailing lists (BugTraq, VulnWatch <http://www.vulnwatch.org/>, comp.risks) to see. You may be surprised.

- *Interposition attacks—DLLs, library paths, client spoofing*: Person-in-the-middle attacks are very popular, mostly because they are very effective. Same goes for PATH hacking, spoofing, and other low-hanging fruit. Carefully consider what happens when an attacker gets between one component and the other components (or between one level of the computing system and the others).

By applying the simple three-step process outlined here, you can greatly improve on a more generic checklist-based approach. There is no substitute for experience and expertise, but as software security knowledge increases, more and more groups should be able to adopt these methods as their own.

Getting Started with Risk Analysis

This whole risk analysis thing seems a bit hard; but risk analysis does not really have to be hard. Sometimes when faced with a seemingly large task like this, it's difficult to get the ball rolling. To counter that problem, Appendix C presents a simple exercise in armchair risk analysis. The idea is to apply some of the ideas you have learned in this chapter to complete a risk analysis exercise on a pretend system (riddled with security flaws). I hope you find the exercise interesting and fun.[13]

Start with something really simple, like the STRIDE model [Howard and LeBlanc 2003]. Develop a simple checklist of attacks and march down the list, thinking about various attack categories (and the related flaws that spawn them) as you go. Checklists are not a complete disaster (as the existence of the attack resistance subprocess shows). In fact, in the hands of an expert, checklists (like the 48 attack patterns in *Exploiting Software* [Hoglund and McGraw 2004]) can be very powerful tools. One problem with checklists is that you are not very likely to find a new, as-yet-to-be-discovered attack if you stick only to the checklist.[14] Another problem is that in the hands of an inexperienced newbie, a checklist is not a very powerful tool. Then again, newbies should not be tasked with architectural risk analysis.

[13]Please try this at home! Hint: Try doing the exercise with a group of friends and a bottle of good wine.

[14]This is important because (smart) attackers use checklists too . . . in order to avoid doing something obvious that will get them caught. On the other hand, script kiddies will bumble right into your defenses, like a roach wandering into a roach motel.

Architectural Risk Analysis Is a Necessity

Risk analysis is, at best, a good general-purpose yardstick by which you can judge the effectiveness of your security design. Since around 50% 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 of any application often yields valuable, business-relevant results. The process of risk analysis identifies system-level vulnerabilities and their probability and impact on the organization. Based on considering the resulting ranked risks, business stakeholders can determine whether to mitigate a particular risk and which control is the most cost effective.