# Secure Software Architecture and Design

## Introduction

### The Critical Role of Architecture and Design

Software architecture and design is where ambiguities and ideas are translated and transformed into reality, where the *what* and *why* of requirements become the *who*, *when*, *where*, and *how* of the software to be. From a functional perspective, this transition from desire to actual form is second only to the requirements phase in contributing to the overall quality and success of the eventual software deliverable. From a security perspective, architecture and design is considered by many experts as the single most critical phase of the SDLC. Good decisions made during this phase will not only yield an approach and structure that are more resilient and resistant to attack, but will often also help to prescribe and guide good decisions in later phases such as code and test. Bad decisions made during this phase can lead to design flaws that can never be overcome or resolved by even the most intelligent and disciplined code and test efforts.

### General Objectives of Software Architecture and Design

- *Completeness*
  - Supports the full scope of the defined requirements
- *Stability*
  - Consistently performs as intended within its defined operational context
- *Flexibility*
  - Can adapt to changing conditions
  - Decomposable such that selected components can be replaced going forward with minimal impact to the software
- *Extensibility*
  - Leverages industry standards
  - Long-lived and resistant to obsolescence
- *Scalability*
  - Operates effectively at any size and load

### Security-Specific Objectives of Software Architecture and Design

- Comprehensive functional security architecture
  - Security features and capabilities are fully enabled
- Attack resistance
  - Contains minimal security weaknesses that could be exploited
- Attack tolerance
  - While resisting attack, software function and capability are not unduly affected
- Attack resilience
  - In the face of successful attack, the effects on the software are minimized

While much of the fanfare of software security today focuses on buffer overflows, SQL injection, and other implementation bugs, the reality is that approximately half of the defects leading to security vulnerabilities found in today's software are actually attributable to flaws in architecture and design [McGraw 2006]. These flaws tend to have a much greater footprint in terms of their exploit and potential security impact within a single piece of software and potentially across multiple projects and systems. The goal of building security into the architecture and design phase of the SDLC is to significantly reduce the number of flaws as early as possible while also minimizing ambiguities and other weaknesses.

# Issues and Challenges

Just as security software (e.g., application firewalls, encryption packages) is not the same thing as software security (the practice of making software more secure), so too security architecture (the architecture of security components) is not the same as secure architecture (architecture that is resilient and resistant to attack). Security is not simply about functionality. Rather, it is about the assurance both that the software will do what it is expected to do and that it will not do what it is not expected to do. The challenge to building security into the architecture and design portion of the SDLC is that not only must the architecture address currently understood security issues—both known weaknesses and attacks—but at its level of abstraction it must also be flexible and resilient under constantly changing security conditions.

This moving target of weakness and vulnerability, combined with the proactive and creative nature of the security attacker, means that no system can ever be perfectly secure. The best that can be achieved is a minimized risk profile accomplished through disciplined and continuous risk management. The practice of architectural risk analysis (involving threat modeling, risk analysis, and risk mitigation planning) performed during the architecture and design phase is one of the cornerstones of this risk management approach. (See Section 7.4.2 for a description of a high-level risk management framework within which to conduct architectural risk analysis.)

Because of their complex yet critical nature, both architectural risk analysis and the basic activities of secure architecture and design require the application of diverse, high-level knowledge. This knowledge is, for the most part, based on experience and historically has been very difficult to come by, leaving this field in the past to become the realm of a small number of experts and gurus. More recently, great strides have been made to capture, codify, and share this sort of knowledge among a much broader audience. With this foundation, every software development team, including architects and designers, can build on the knowledge of veteran experts. Some examples of these knowledge resources (security principles, security guidelines, and attack patterns) are described in Section 4.3.

This chapter introduces some of the resources available, in the form of practices and knowledge, for building security into the architecture and design phase of the SDLC. Although it is not a complete treatment of the topic, it does provide the more critical resources needed to address security as part of software development processes. Based on the context of your project, you can decide how best to integrate these practices and knowledge resources into your processes.

| **Specific Project Manager Concerns During Software Architecture and Design** | |
|---|---|
| ***Concern*** | ***Process/Knowledge*** |
| Delivering what was specified<br>• Deliverables must fulfill the objectives of the project | Architectural risk analysis |
| Getting it right the first time<br>• Minimize rework | Architectural risk analysis |
| Effectively shoring up staff expertise shortfalls | Security principles<br>Security guidelines<br>Attack patterns |

## Software Security Practices for Architecture and Design: Architectural Risk Analysis

If you are looking to integrate security concerns into the software architecture and design phase of the SDLC, the practice of architectural risk analysis is of utmost importance. Architectural risk analysis is intended to provide assurance that architecture and design-level security concerns are identified and addressed as early as possible in the life cycle, yielding improved levels of attack resistance, tolerance, and resilience.[3] Without this kind of analysis, architectural flaws will remain unaddressed throughout the life cycle (though they often cause trouble during implementation and testing) and will likely result in serious security vulnerabilities in the deployed software. No other single action, practice, or resource applied during the architecture and design phase of the SDLC will have as much positive impact on the security risk profile of the software being developed.

While architectural risk analysis does not focus primarily on assets, it does depend on the accurate identification of the software's ultimate purpose and understanding of how that purpose ties into the business's activities to qualify and quantify the risks identified during this process. For this reason, a solid understanding of the assets that the software guards or uses should be considered a prerequisite to performing architectural risk analysis. Methodologies for asset identification are available from a wide variety of risk management sources.

During risk analysis, potential threats are identified and mapped to the risks they bring to bear. These risks are a function of the likelihood of a given threat exploiting a particular potential vulnerability and the resulting impact of that adverse event on the organization or on information assets. To determine the likelihood of an adverse event, threats to the software must be analyzed in conjunction with the potential vulnerabilities and the security controls in place for the software. The impact refers to the magnitude of harm that could be caused by realized risk. Its level is governed by the potential costs and losses to individuals or to the organization, its mission, or its assets and, in turn, leads to assignment of a relative value to the information assets and resources affected (e.g., the criticality and sensitivity of the software components and data). In the end, the results of the risk analysis help identify appropriate controls, revisions, or actions for reducing or eliminating risk during the risk mitigation process.

The risk analysis methodology consists of **six activities**:

- **Software characterization**
- **Threat analysis**
- **Architectural vulnerability assessment**
- **Risk likelihood determination**
- **Risk impact determination**
- **Risk mitigation planning**

These activities are described next.

# Software Characterization

The first step required in analyzing any software, whether new or existing, for risk is to achieve a full understanding of what the software is and how it works. For architectural risk analysis, this understanding requires at least minimal description using high-level diagramming techniques. The exact format used may vary from organization to organization and is not critically important. What *is* important is coming up with a comprehensive, yet concise picture that unambiguously illustrates the true nature of the software.

One format that has proven itself particularly effective for this purpose is a simple whiteboard-type, high-level, one-page diagram that illustrates how the components are wired together, as well as how control and data flow are managed [McGraw 2006]. This "forest-level" view is crucial for identifying architecture and design-level flaws that just don't show up during code-level reviews. For more detailed analysis, this one-page diagram can be fleshed out as necessary with more detailed design descriptions.

Gathering information for this characterization of the software typically involves reviewing a broad spectrum of system artifacts and conducting in-depth interviews with key high-level stakeholders such as product/program managers and software architects. Useful artifacts to review for software characterization include, but are not limited to, the following items:

- Software business case
- Functional and nonfunctional requirements
- Enterprise architecture requirements
- Use case documents
- Misuse/abuse case documents
- Software architecture documents describing logical, physical, and process views
- Data architecture documents
- Detailed design documents such as UML diagrams that show behavioral and structural aspects of the system
- Software development plan
- Transactions security architecture documents
- Identity services and management architecture documents
- Quality assurance plan
- Test plan
- Risk management plan
- Software acceptance plan
- Problem resolution plan

- Configuration and change management plan

In cases where the software is already in production or uses resources that are in production (e.g., databases, servers, identity systems), these systems may have already been audited and assessed. These assessments, when they exist, may provide a rich set of analysis information.

Although it is often not practical to model and depict all possible interrelationships, the goal of the software characterization activity is to produce one or more documents that depict the vital relationships between critical parts of the software. Using information gathered through asset identification, interviews, and artifact analysis, the diagrams and documents gradually take shape.

Figure 4–1 presents an example of a high-level, one-page system software architecture diagram. This diagram shows major system components, their interactions, and various zones of trust.[4] Avatars and their associated arrows represent potential attackers and attack vectors against the system. These potential threats and attack vectors are further fleshed out and detailed during the following stages of architectural risk analysis.

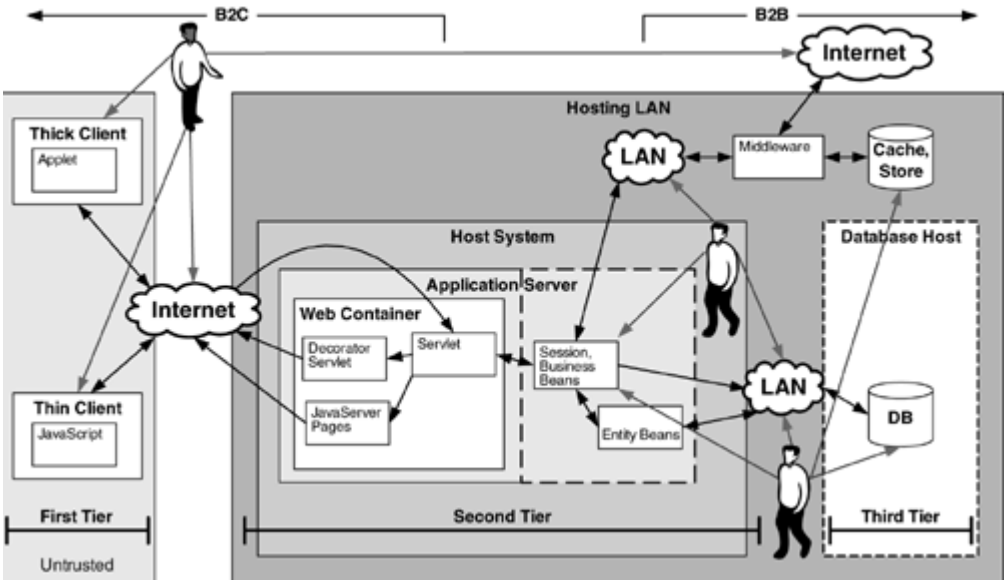


**Figure 4–1. High-level, one-page system software architecture diagram**

# Threat Analysis

Threats are agents that violate the protection of information assets and site security policy. Threat analysis identifies relevant threats for a specific architecture, functionality, and configuration. It may assume a given level of access and skill level that the attacker may possess. During this analysis, threats may be mapped to vulnerabilities to understand how the software may be exploited. A mitigation plan is composed of countermeasures that are considered to be effective against the identified vulnerabilities that these threats exploit.

Attackers who are not technologically sophisticated are increasingly performing attacks on software without really understanding what it is they are exploiting, because the weakness was discovered by someone else. These individuals, who are often referred to as “script kiddies,” typically do not launch attacks in an effort to obtain specific information or target specific

organizations. Instead, they use the knowledge of various vulnerabilities to broadly scan the entire Internet for systems that possess those vulnerabilities, and then attack whichever ones they come across. At the other end of the attacker spectrum, highly skilled threats targeting very specific organizations, systems, and assets have become increasing prevalent. Threat analysis should evaluate and identify threats across this spectrum.

Table 4–1, which was developed by NIST, summarizes a very generic set of potential threat sources [NIST 2002, p. 14].

**Table 4–1. NIST Threat Identification and Characterization**

| ***Threat Source*** | ***Motivation*** | ***Threat Actions*** |
|---|---|---|
| Cracker | Challenge<br><br>Ego<br><br>Rebellion | • System profiling<br>• Social engineering<br>• System intrusion, break-ins<br>• Unauthorized system access |
| Computer criminal | Destruction of information<br><br>Illegal information disclosure<br><br>Monetary gain<br><br>Unauthorized data alteration | • Computer crime (e.g., cyberstalking)<br>• Fraudulent act (e.g., replay, impersonation, interception)<br>• Information bribery<br>• Spoofing<br>• System intrusion<br>• Botnets<br>• Malware: Trojan horse, virus, worm, spyware<br>• Spam<br>• Phishing |
| Terrorist | Blackmail<br><br>Destruction<br><br>Exploitation<br><br>Revenge<br><br>Monetary gain<br><br>Political gain | • Bomb<br>• Information warfare<br>• System attack (e.g., distributed denial of service)<br>• System penetration<br>• System tampering |
| Industrial espionage | Competitive advantage<br><br>Economic espionage<br><br>Blackmail | • Economic exploitation<br>• Information theft<br>• Intrusion on personal privacy<br>• Social engineering<br>• System penetration<br>• Unauthorized system access (access to classified, proprietary, and/or |

| *Threat Source* | *Motivation* | *Threat Actions* |
|---|---|---|
| | | technology-related information) |
| Insiders (poorly trained, disgruntled, malicious, negligent, dishonest, or terminated employees) [CERT 2007] | Curiosity<br>Ego<br>Intelligence<br>Monetary gain<br>Revenge<br>Unintentional errors and omissions (e.g., data entry errors, programming errors)<br>Wanting to help the company (victims of social engineering)<br>Lack of procedures or training | • Assault on an employee<br>• Blackmail<br>• Browsing of proprietary information<br>• Computer abuse<br>• Fraud and theft<br>• Information bribery<br>• Input of falsified, corrupted data<br>• Interception<br>• Malicious code (e.g., virus, logic bomb, Trojan horse)<br>• Sale of personal information<br>• System bugs<br>• System intrusion<br>• System sabotage<br>• Unauthorized system access |

An issue that greatly complicates the prevention of threat actions is that the attacker's underlying intention often cannot be determined. Both internal and external threat sources may exist, and an attack taxonomy should consider the motivation and capability of both types of threats. Internal attacks might be executed by individuals such as disgruntled employees and contractors. It is important to note that nonmalicious use by threat actors may also result in software vulnerabilities being exploited. Internal threat actors may act either on their own or under the direction of an external threat source (for example, an employee might install a screensaver that contains a Trojan horse).

Some threat actors are external. These attackers could include structured external, transnational external, and unstructured external threats:

- Structured external threats are generated by a state-sponsored entity, such as a foreign intelligence service. The resources supporting the structured external threat are usually quite substantial and highly sophisticated.
- Transnational threats are generated by organized nonstate entities, such as drug cartels, crime syndicates, and terrorist organizations. Such threats generally do not have as many resources behind them as do structured threats (although some of the larger transnational threat organizations may have more resources than some smaller, structured threat organizations). The nature of the transnational external threat makes it more difficult to trace and provide a response, however. These kinds of threats can target members or staff of the Treasury, for example, by employing any or all of the techniques mentioned above.

- Unstructured external threats are usually generated by individuals such as crackers. Threats from this source typically lack the resources of either structured or transnational external threats but nonetheless may be very sophisticated. The motivation of such attackers is generally—but not always—less hostile than that underlying the other two classes of external threats. Unstructured threat sources generally limit their attacks to information system targets and employ computer attack techniques. New forms of loosely organized virtual hacker organizations (*hacktivists*—hackers and activists) are also emerging.

# Architectural Vulnerability Assessment

Vulnerability assessment examines the preconditions that must be present for vulnerabilities to be exploited and assesses the states that the software might enter upon exploit. Three activities make up architectural vulnerability assessment: attack resistance analysis, ambiguity analysis, and dependency analysis. As with any quality assurance process, risk analysis testing can prove only the presence—not the absence—of flaws. Architectural risk analysis studies vulnerabilities and threats that might be malicious or nonmalicious in nature. Whether the vulnerabilities are exploited intentionally (malicious) or unintentionally (nonmalicious), the net result is that the desired security properties of the software may be affected.

One advantage when conducting vulnerability assessment at the architectural level is the ability to see the relationships and effects at a "forest-level" rather than "tree-level" view. As described earlier, this perspective takes the form of a one-page overview of the system created during software characterization. In practice, it means assessing vulnerabilities not just at a component or function level, but also at interaction points. Use-case models help to illustrate the various relationships among system components. In turn, the architecture risk analysis should factor these relationships into the vulnerabilities analysis and consider vulnerabilities that may emerge from these combinations.

## Attack Resistance Analysis

Attack resistance analysis is the process of examining software architecture and design for common weaknesses that may lead to vulnerabilities and increase the system's susceptibility to common attack patterns.

Many known weaknesses are documented in the software security literature, ranging from the obvious (failure to authenticate) to the subtle (symmetric key management problems). Static code checkers, runtime code checkers, profiling tools, penetration testing tools, stress test tools, and application scanning tools can find some security bugs in code, but they do not as a rule address architectural flaws. For example, a static code checker can flag bugs such as buffer overflows, but it cannot identify security vulnerabilities such as transitive trust mistakes.[5] Architectural-level flaws can currently be found only through human analysis.

When performing attack resistance analysis, consider the architecture as it has been described in the artifacts that were reviewed for asset identification. Compare it against a body of known bad practices, such as those outlined in the Common Weakness Enumeration (CWE) [CWE 2007], or known good principles, such as those outlined in Section 4.3. For example, the *principle of least privilege* dictates that all software operations should be performed with the least possible privilege required to meet the need. To consider architecture in light of this

principle, find all the areas in the software that operate at an elevated privilege level. To do so, you might perhaps diagram the system's major modules, classes, or subsystems and circle areas of high privilege versus areas of low privilege. Consider the boundaries between these areas and the kinds of communications that occur across those boundaries.

Once potential vulnerabilities have been identified, the architecture should be assessed for how well it would fare against common attack patterns such as those outlined in the Common Attack Pattern Enumeration and Classification (CAPEC) [CAPEC 2007]. CAPEC describes the following classes of attack, among others:

- Abuse of functionality
- Spoofing
- Probabilistic techniques
- Exploitation of privilege or trust
- Injection
- Resource manipulation
- Time and state attacks

Relevant attack patterns should be mapped against the architecture, with special consideration being given to areas of identified vulnerability. Any attack found to be viable against identified vulnerabilities should be captured and quantified as a risk to the software.

## Ambiguity Analysis

Ambiguity is a rich source of vulnerabilities when it exists between requirements or specifications and development. A key role of architecture and design is to eliminate the potential misunderstandings between business requirements for software and the developers' implementation of the software's actions. All artifacts defining the software's function, structure, properties, and policies should be examined for any ambiguities in description that could potentially lead to multiple interpretations. Any such opportunities for multiple interpretations constitute a risk to the software.

A key consideration is to note places where the requirements or architecture are ambiguously defined or where the implementation and architecture either disagree or fail to resolve the ambiguity. For example, a requirement for a Web application might state that an administrator can lock an account, such that the user can no longer log in while the account remains locked. But what about sessions for that user that are actively in use when the administrator locks the account? Is the user suddenly and forcibly logged out, or does the active session remain valid until the user logs out? In an existing system, the authentication and authorization architecture must be compared to the actual implementation to learn the answer to this question. The security ramifications of logins that persist even after the account is locked should be balanced against the sensitivity of the information assets being guarded.

## Dependency Analysis

An architectural risk assessment must include an analysis of the vulnerabilities associated with the software's execution environment. The issues addressed as part of this assessment will include operating system vulnerabilities, network vulnerabilities, platform vulnerabilities (popular platforms include WebLogic, WebSphere, PHP, ASP.net, and Jakarta), and interaction vulnerabilities resulting from the interaction of components. The goal of this

analysis is to develop a list of software or system vulnerabilities that could be accidentally triggered or intentionally exploited, resulting in a security breach or a violation of the system's security policy. When credible threats can be combined with the vulnerabilities uncovered in this exercise, a risk exists that needs further analysis and mitigation.

The types of vulnerabilities that will exist and the methodology needed to determine whether the vulnerabilities are present can vary. At various times, the analysis might focus on the organization's security policies, planned security procedures, nonfunctional requirement definitions, use cases, misuse/abuse cases, architectural platforms/components/services, and software security features and security controls (both technical and procedural) used to protect the system, among other issues.

Independent of the life-cycle phase, online vulnerability references should be consulted. Numerous such resources are available, including the National Vulnerability Database (NVD) [NIST 2007], the Common Vulnerabilities and Exposures (CVE) database [CVE 2007], and the BugTraq email list [SecurityFocus 2007], among others. These sites and lists should be consulted regularly to keep the vulnerability list up-to-date for a given architecture.

## Vulnerability Classification

Classifying vulnerabilities allows for pattern recognition of vulnerability types. This exercise, in turn, may enable the software development team to recognize and develop countermeasures to deal with classes of vulnerabilities by dealing with the vulnerabilities at a higher level of abstraction. The most comprehensive and mature example of such a classification taxonomy currently available is the Common Weakness Enumeration (CWE) [CWE 2007], which has been crafted as a normalized aggregation of dozens of the other such taxonomies recognized and respected by the industry. The CWE includes seven top-level categories for architecture and source code [Tsipenyuk 2005]:

- Data Handling
- API Abuse
- Security Features
- Time and State
- Error Handling
- Code Quality
- Encapsulation

## Mapping Threats and Vulnerabilities

The combination of threats and vulnerabilities illustrates the risks to which the software is exposed. Several models exist to categorize the areas where these threats and vulnerabilities frequently intersect. One example is Microsoft's STRIDE [Howard 2002], which provides a model of risks to a computer system related to spoofing, tampering, repudiation, information disclosure, denial of service, and elevation of privilege.

Risk classification assists in communicating and documenting risk management decisions. Risk mitigation mechanisms should map to the risk category or categories of the threats and vulnerabilities that have been identified through this effort.

# Risk Likelihood Determination

Having determined which threats are important and which vulnerabilities might exist to be exploited, it can be useful to estimate the likelihood of the various possible risks. In software security, "likelihood" is a qualitative estimate of how likely a successful attack will be, based on analysis and past experience. Because of the complexity of the software domain and the number of variables involved in risk analysis, this likelihood measure is not an actual mathematical *probability* of a successful attack. Nonetheless, the concept of likelihood can be useful when prioritizing risks and evaluating the effectiveness of potential mitigations.

Consider these factors, all of which are incorporated in the likelihood estimation:

- The threat's motivation and capability
- The vulnerability's impact (and therefore attractiveness to an attacker)
- The effectiveness of current controls

Threats' motivation and capability will vary widely. For instance, a college student who hacks for fun is less highly motivated than a paid hacker who has backing or the promise of a significant payment from an organized crime cabal. A former employee who has a specific grievance against a company will be more motivated and better informed than an outsider who has no special knowledge of the target software's internal workings.

The effectiveness of current controls characterizes how high the bar is set for an intentional attacker or how unlikely an accidental failure is. For example, simple user IDs and passwords can be compromised much more easily than most two-factor authentication systems. Adding a second authentication factor raises the bar for a would-be threat. However, if the second factor in the authentication is a biometric thumbprint reader that can be spoofed with latent image recovery techniques, then the system may be less secure than desired.

The likelihood estimation is a subjective combination of these three qualities—motivation, directness of vulnerability, and compensating controls—and is typically expressed as a rating of high, medium, or low. Table 4–2 describes one potential model for calculating likelihood from these three qualities. In most cases, such hard-and-fast rules cannot adequately cover all contingencies, so calculations must be tailored for each context and remain flexible to interpretation.

**Table 4–2. Model for Calculating Likelihood**

| | |
|---|---|
| High | The three qualities are all weak. A threat is highly motivated and sufficiently capable, a vulnerability exists that is severe, and controls to prevent the vulnerability from being exploited are ineffective. |
| Medium | One of the three qualities is compensating, but the others are not. The threat is perhaps not very motivated or not sufficiently capable, the controls in place might be reasonably strong, or the vulnerability might be not very severe. |
| Low | Two or more of the three qualities are compensating. The threat might lack motivation or capability; strong controls might be in place to prevent, or at least significantly impede, the vulnerability from being exploited; and the vulnerability might have a very low impact. |

# Risk Impact Determination

Independent of the risk's likelihood and the system's controls against it, the risk's impact must be determined. That is, what consequences will the business face if the worst-case scenario in the risk description comes to pass? Furthermore, the risk analysis must account for other credible scenarios that are not the worst case, yet are bad enough to warrant attention. This section discusses three aspects of risk impact determination: identifying the threatened assets, identifying business impact, and determining impact locality.

## Identify Threatened Assets

The assets threatened by realization of the risk, and the nature of what will happen to them, must be identified. Common impacts on information assets include loss of data, corruption of data, unauthorized or unaudited modification of data, unavailability of data, corruption of audit trails, and insertion of invalid data.

## Identify Business Impact

The business will suffer some impact if an attack takes place. It is of paramount importance to characterize those effects in as specific terms as possible. Risk management efforts are almost always funded ultimately by management in the organization whose primary concern is monetary. Those managers' support and understanding can be assured only by quantifying software risks in terms of their fiscal implications. If the worst possible consequence of a software failure is the loss of $10,000 to the business, but it will take $20,000 in labor-hours and testing to fix the software, the return on the mitigation investment does not make financial sense. Furthermore, correct financial assessment of the risk's effects drives prioritization. It is usually more important to fix a flaw that can precipitate a $25 million drop in the company's market capitalization than to address a flaw that can expose the business to a regulatory penalty of $500,000. Unless software risks are tied to business impacts, however, such reasoning is not possible.

Examples of business impacts include loss of market share, loss of reputation, depreciation of stock value, fines, legal fees and judgments, costs of technical remediation, and theft. A good example of a case in which all of these impacts are relevant is the TJX data breach, where lax wireless security led to large quantities of customer data being accessed through exploitation of a vulnerability. TJX suffered severe brand damage and costs that some analysts predict may reach into the billions of dollars:

IPLocks, a compliance and database security company, released a report earlier this month estimating that the TJX breach will eventually cost the company $100 per lost record, or a total of $4.5 billion. The company based the estimate on the accumulated costs of fines, legal fees, notification expenses, and brand impairment, according to Adrian Lane, the company's chief technology officer [Gaudin 2007].

## Risk Exposure Statement

The risk exposure statement combines the likelihood of a risk with the impact of that risk. The product of these two analyses provides the overall summary of risk exposure for the

organization for each risk. Table 4–3 describes a method of generating the risk exposure statement.

| | | *Impact* | | |
|---|---|---|---|---|
| | | *Low* | *Medium* | *High* |
| *Likelihood* | *Low* | Low | Low | Medium |
| | *Medium* | Low | Medium | High |
| | *High* | Medium | High | High |

**Figure 4–3. Risk Exposure Calculation Matrix**

The risk exposure statement generalizes the overall exposure of the organization relative to the given risk and offers more granular visibility to both its impact and its likelihood. In this way, the risk exposure statement gives the organization finer-grained control over risk management but does not require all risks to be eliminated. As Alan Greenspan, Chairman of the Federal Reserve Board, said in 1994:

There are some who would argue that the role of the bank supervisor is to minimize or even eliminate bank failure, but this view is mistaken in my judgment. The willingness to take risk is essential to the growth of the free market economy . . . [i]f all savers and their financial intermediaries invested in only risk-free assets, the potential for business growth would never be realized.[6]

# Risk Mitigation Planning

Mitigation of a risk entails changing the architecture of the software or the business in one or more ways to reduce the likelihood or the impact of the risk. Formal and informal testing, such as penetration testing, may be used to test the effectiveness of these mitigation measures.[7]

Mitigations aimed at architectural flaws are often more complicated to implement than mitigations focusing on coding bugs, which tend to be more localized. Architectural mitigations often require changes to multiple modules, multiple systems, or at least multiple classes; and the affected entities may be managed and implemented by different teams. Thus, when a flaw is found, the fix often requires agreement across multiple teams, testing of multiple integrated modules, and synchronization of release cycles that may not always be present in the different modules.

Measures intended to reduce the likelihood of a risk can take several forms. Raising the bar in terms of the skills necessary to exploit a vulnerability is often a first step in this direction. For example, changing authentication mechanisms from user IDs and passwords to pre-shared public key certificates can make it far more difficult to impersonate a user. Reducing the period of time when a vulnerability is available for exploit is another way to reduce the likelihood of a risk coming to fruition. For example, if sessions expire after 10 minutes of inactivity, then the window of opportunity for session hijacking is about 10 minutes long.

Measures intended to reduce the impact of a risk can also take several forms. Most developers immediately consider eliminating the vulnerability altogether or fixing the flaw so that the architecture cannot be exploited. Cryptography can help, for example, as long as it is applied correctly. It is easier to detect corruption in encrypted data than in unencrypted data, and encrypted data is more difficult for attackers to use if they collect it. Sometimes, from a business point of view, it makes more sense to focus on building functionality to detect and log successful exploits and providing enough related auditing information to effectively recover after the fact. Remediating a broken system might be too expensive, whereas adding enough functionality to allow recovery after an exploit might be sufficient.

Many mitigations can be described as either detection strategies or correction strategies. Depending on the cost of making failure impossible through correction, the organization may find it much more cost-effective to enable systems to detect and repair failure quickly and accurately. Imagine a software module that is very temperamental and tends to crash when provided with bad input and (for the sake of argument) cannot be modified or replaced. A focus on correction would add business logic to validate input and make sure that the software module never received input that it could not handle. In contrast, a focus on detection would add monitoring or other software to watch for a crash of the module and try to restart the module quickly with minimal impact.

Mitigation is never without cost. The fact that remediating a problem costs money makes it even more important to handle the risk impact determination step well. It is typically straightforward to characterize the cost of implementing mitigations—for example, in terms of hours of labor, cost of shipping new units with the improved software, or delay entering the market with new features because old ones must be fixed. This ability to characterize the mitigation's cost, however, is of little value unless the cost of the risk's business impact is known.

Of course, risk mitigation mechanisms themselves can introduce threats and vulnerabilities to the software. Designs also evolve and change over time. The risk analysis process is therefore iterative, accounting for and guarding against new risks that might have been introduced.

# Recapping Architectural Risk Analysis

Architectural risk analysis is a critical activity in assuring the security of software. Current technologies cannot automate the detection process for architectural flaws, which often prove to be among the most damaging security defects found in software. Human-executed architectural risk analysis is the only way to effectively address these problems.

Risk management is an ongoing process, and architectural risk analysis follows suit. Architectural risk analysis is conducted at discrete points in time and performed iteratively as an ongoing process across the life cycle. As the software evolves, its architecture must be kept up-to-date. The body of known attack patterns (discussed in Chapter 2) is always growing, so continued success in attack resistance analysis depends on remaining current in software security trends. Ambiguity analysis is always necessary as conditions and the software evolve. Even with that focus, it is worthwhile to occasionally step back and reappraise the entire system for ambiguity. As platforms are upgraded and evolve in new directions, each subsequent release will fix older problems—and probably introduce new ones. Given this probability, dependency analysis must continue throughout the life of the product.

A master list of risks should be maintained during all stages of the architectural risk analysis. This list should be continually revisited to determine mitigation progress for the project at hand and to help improve processes for future projects. For example, the number of risks identified in various software artifacts and/or software life-cycle phases may be used to identify problematic areas in the software development process. Likewise, the number of risks mitigated over time may be used to show concrete progress as risk mitigation activities unfold. Ideally, the display and reporting of risk information should be aggregated in some automated way and displayed in a risk "dashboard" that enables the development team to make accurate and informed decisions.

# Software Security Knowledge for Architecture and Design:

**Security Principles, Security Guidelines, and Attack Patterns**

One of the significant challenges you may face when seeking to integrate security into the architecture and design of your software projects is the scarcity of experienced architects who have a solid grasp of security concerns. The problem here is the learning curve associated with security concerns: Most developers simply don't have the benefit of the years and years of lessons learned that an expert in software security can call on. To help address this issue, you can leverage codified knowledge resources—such as security principles, security guidelines, and attack patterns—to bolster your software architects' basic understanding of software security. These knowledge resources serve as the fuel that effectively drives the process of architectural risk analysis. They guide the architects and designers by suggesting which questions to ask, which issues to consider, and which mitigation measures to pursue. Although projects should still enlist the services of at least one truly experienced software security architect, knowledge resources such as security principles, security guidelines, and attack patterns can help organizations to more effectively distribute these scarce resources across projects.

## Security Principles

Security principles are a set of high-level practices derived from real-world experience that can help guide software developers (software architects and designers in particular) in building more secure software. Jerome Saltzer and Michael Schroeder were the first researchers to correlate and aggregate high-level security principles in the context of protection mechanisms [Saltzer 1975]. Their work provides the foundation needed for designing and implementing secure software systems. Principles define effective practices that are applicable primarily to architectural-level software decisions and are recommended regardless of the platform or language of the software. As with many architectural decisions, principles, which do not necessarily guarantee security, at times may exist in opposition to each other, such that appropriate tradeoffs must be made. Software architects, whether they are crafting new software or evaluating and assessing existing software, should always apply these design principles as a guide and yardstick for making their software more secure.

Despite the broad misuse of the term "principle," the set of things that should be considered security principles is actually very limited. The Principles content area of the Build Security In (BSI) Web site [BSI 17] presents an aggregation of such principles, including many taken from Saltzer and Schroeder's original work and a few offered by other thought leaders such as Gary

McGraw, Matt Bishop, Mike Howard, David LeBlanc, Bruce Schneier, John Viega, and NIST. The filter applied to decide what constitutes a principle and what does not is fairly narrow, recognizing that such lasting principles do not come along every day. The high-level and lasting nature of these principles leads to their widespread recognition, but also to a diversity of perspectives and interpretation.[9]

By leveraging security principles, a software development team can benefit from the guidance of the industry's leading practitioners and can learn to ask the right questions of their software architecture and design so as to avoid the most prevalent and serious flaws. Without these security principles, the team is reduced to relying on the individual security knowledge of its most experienced members.

The following list outlines a core set of security principles that every software development team member—from the people writing code on up to the project managers—should be aware of and familiar with. While this information is most actively put into play by software architects and designers, awareness of these foundational concerns across the team is a powerful force for reducing the risk to the software posed by security issues. Brief descriptions are given here for each principle; more detailed descriptions and examples are available on the Principles content area on the BSI Web site.

**The Principles for Software Security**

- Least privilege
- Failing securely
- Securing the weakest link
- Defense in depth
- Separation of privilege
- Economy of mechanism
- Least common mechanism
- Reluctance to trust
- Never assuming that your secrets are safe
- Complete mediation
- Psychological acceptability
- Promoting privacy

**The Principle of Least Privilege**

Only the minimum necessary rights should be assigned to a subject[10] that requests access to a resource and should be in effect for the shortest duration necessary (remember to relinquish privileges). Granting permissions to a user beyond the scope of the necessary rights of an action can allow that user to obtain or change information in unwanted ways. In short, careful delegation of access rights can limit attackers' ability to damage a system.

**The Principle of Failing Securely**

When a system fails, it should do so securely. This behavior typically includes several elements: secure defaults (the default is to deny access); on failure, undo changes and restore the system to a secure state; always check return values for failure; and in conditional code/filters, make sure that a default case is present that does the right thing. The confidentiality and integrity of a system should remain unbreached even though availability has been lost.

During a failure, attackers must not be permitted to gain access rights to privileged objects that are normally inaccessible. Upon failing, a system that reveals sensitive information about the failure to potential attackers could supply additional knowledge that threat actors could then use to create a subsequent attack. Determine what may occur when a system fails and be sure it does not threaten the system.

**The Principle of Securing the Weakest Link**

Attackers are more likely to attack a weak spot in a software system than to penetrate a heavily fortified component. For example, some cryptographic algorithms can take many years to break, so attackers are unlikely to attack encrypted information communicated in a network. Instead, the endpoints of communication (e.g., servers) may be much easier to attack. Knowing when the weak spots of a software application have been fortified can indicate to a software vendor whether the application is secure enough to be released.

**The Principle of Defense in Depth**

Layering security defenses in an application can reduce the chance of a successful attack. Incorporating redundant security mechanisms requires an attacker to circumvent each mechanism to gain access to a digital asset. For example, a software system with authentication checks may prevent intrusion by an attacker who has subverted a firewall. Defending an application with multiple layers can eliminate the existence of a single point of failure that compromises the security of the application.

**The Principle of Separation of Privilege**

A system should ensure that multiple conditions are met before it grants permissions to an object. Checking access on only one condition may not be adequate for enforcing strong security. If an attacker is able to obtain one privilege but not a second, the attacker may not be able to launch a successful attack. If a software system largely consists of one component, however, it will not be able to implement a scheme of having multiple checks to access different components. Compartmentalizing software into separate components that require multiple checks for access can inhibit an attack or potentially prevent an attacker from taking over an entire system.

**The Principle of Economy of Mechanism**

One factor in evaluating a system's security is its complexity. If the design, implementation, or security mechanisms are highly complex, then the likelihood that security vulnerabilities will exist within the system increases. Subtle problems in complex systems may be difficult to find, especially in copious amounts of code. For instance, analyzing the source code that is responsible for the normal execution of a functionality can be a difficult task, but checking for alternative behaviors in the remaining code that can achieve the same functionality may prove even more difficult. Simplifying design or code is not always easy, but developers should strive for implementing simpler systems when possible.

**The Principle of Least Common Mechanism**

Avoid having multiple subjects share those mechanisms that grant access to a resource. For example, serving an application on the Internet allows both attackers and users to gain access

to the application. In this case, sensitive information might potentially be shared between the subjects via the same mechanism. A different mechanism (or instantiation of a mechanism) for each subject or class of subjects can provide flexibility of access control among various users and prevent potential security violations that would otherwise occur if only one mechanism were implemented.

**The Principle of Reluctance to Trust**

Developers should assume that the environment in which their system resides is insecure. Trust—whether it is extended to external systems, code, or people—should always be closely held and never loosely given. When building an application, software engineers should anticipate malformed input from unknown users. Even if users are known, they are susceptible to social engineering attacks, making them potential threats to a system. Also, no system is ever 100 percent secure, so the interface between two systems should be secured. Minimizing the trust in other systems can increase the security of your application.

**The Principle of Never Assuming That Your Secrets Are Safe**

Relying on an obscure design or implementation does not guarantee that a system is secure. You should always assume that an attacker can obtain enough information about your system to launch an attack. For example, tools such as decompilers and disassemblers may allow attackers to obtain sensitive information that may be stored in binary files. Also, insider attacks, which may be accidental or malicious, can lead to security exploits. Using real protection mechanisms to secure sensitive information should be the ultimate means of protecting your secrets.

**The Principle of Complete Mediation**

A software system that requires access checks to an object each time a subject requests access, especially for security-critical objects, decreases the chances that the system will mistakenly give elevated permissions to that subject. By contrast, a system that checks the subject's permissions to an object only once can invite attackers to exploit that system. If the access control rights of a subject are decreased after the first time the rights are granted and the system does not check the next access to that object, then a permissions violation can occur. Caching permissions can increase the performance of a system, albeit at the cost of allowing secured objects to be accessed.

**The Principle of Psychological Acceptability**

Accessibility to resources should not be inhibited by security mechanisms. If security mechanisms hinder the usability or accessibility of resources, then users may opt to turn off those mechanisms. Where possible, security mechanisms should be transparent to the users of the system or, at most, introduce minimal obstruction. Security mechanisms should be user friendly to facilitate their use and understanding in a software application.

**The Principle of Promoting Privacy**

Protecting software systems from attackers who may obtain private information is an important part of software security. If an attacker breaks into a software system and steals private information about a vendor's customers, then those customers may lose their confidence in the

software system. Attackers may also target sensitive system information that can supply them with the details needed to attack that system. Preventing attackers from accessing private information or obscuring that information can alleviate the risk of information leakage.

### Recapping Security Principles

Security principles are the foundational tenets of the software security domain. They are long-standing, universal statements of how to build software the right way if security is a concern (which it always is). These principles represent the experiential knowledge of the most respected thought leaders and practitioners in the field of software security. By leveraging them, your team gains access to the scalable wisdom necessary for assessing and mitigating the security risk posed by your software's architecture and design.

For a full understanding and treatment of the security principles discussed here, we recommend that you review the more detailed content available on the Build Security In Web site [BSI 17].

## Security Guidelines

Like security principles, security guidelines are an excellent resource to leverage in your projects. They represent experiential knowledge gained by experts over many years of dealing with software security concerns within more specific technology contexts than those covered by the broader security principles. Such guidelines can inform architectural decisions and analysis, but they also represent an excellent starting point and checklist for software designers who are charged with the task of integrating security concerns into their design efforts. You should ensure that resources such as security guidelines are readily available and frequently consulted by the software designers on your teams before, during, and after the actual execution of the architecture and design phase of the SDLC.

### What Do Security Guidelines Look Like?

Numerous interpretations of what security guidelines are and what they look like have been put forth. The BSI Web site contains one interpretation. Gary McGraw's book *Software Security: Building Security In* [McGraw 2006] presents another.[12]

The full description for one of the security guidelines from the BSI Web site is included here as an example. This guideline is not necessarily any more important than any other guideline or interpretation; it was chosen at random to demonstrate the value of this resource.

#### Guideline: Follow the Rules Regarding Concurrency Management

Failure to follow proper concurrency management protocols can introduce serious vulnerabilities into a system. In particular, concurrent access to shared resources without using appropriate concurrency management mechanisms produces hard-to-find vulnerabilities. Many "functions" that are necessary to use can introduce "time of check/time of use" vulnerabilities [Viega 2001].

When multiple threads of control attempt to share the same resource but do not follow the appropriate concurrency protection protocol, then any of the following results are possible:

- **Deadlock:** One or more threads may become permanently blocked [Johansson 2005].
- **Loss of information:** Saved information is overwritten by another thread [Gong 2003; Pugh 1999; Manson 2001, 2005].
- **Loss of integrity of information:** Information written by multiple threads may be arbitrarily interlaced [Gong 2003; Pugh 1999; Manson 2001, 2005].
- **Loss of liveness:** Imbalance in access to shared resources by competing threads can cause performance problems [Gong 2003; Pugh 1999; Manson 2001, 2005].

Any of these outcomes can have security implications, which are sometimes manifested as apparent logic errors (decisions made based on corrupt data).

#### Competing "Systems" (Time of Check/Time of Use)

This is the most frequently encountered subclass of concurrency-related vulnerabilities. Many of the defects that produce these vulnerabilities are unavoidable owing to limitations of the execution environment (i.e., the absence of proper concurrency control mechanisms). A common mitigation tactic is to minimize the time interval between check and use. An even more effective tactic is to use a "check, use, check" pattern that can often detect concurrency violations, though not prevent them.

Applicable Context

All of the following must be true:

- Multiple "systems" must be operating concurrently.
- At least two of those systems must use a shared resource (e.g., file, device, database table row).
- At least one of those systems must use the shared resource in any of the following ways:
    - Without using any concurrency control mechanism. This includes the situation where no such mechanism exists, such as conventional UNIX file systems, causing corruption or confusion.
    - Using the right concurrency control mechanism incorrectly. This includes situations such as not using a consistent resource locking order across all systems (e.g., in databases), causing deadlocks.
    - Using the wrong concurrency control mechanism (even if it used correctly). This includes situations where a give resource may support multiple concurrency control mechanisms that are independent of one another [e.g., UNIX lockf() and flock()], causing corruption or confusion.

These defects are frequently referred to as time of check/time of use (or TOCTOU) defects because APIs providing access to the resource neither provide any concurrency control operations nor perform any implicit concurrency control. In this case, a particular condition (e.g., availability of resource, resource attributes) is checked at one point in time and later program actions are based on the result of that check, but the condition could change at any time, because no concurrency control mechanism guarantees that the condition did not change.

#### Competing Threads within a "System" (Races)

The second largest class of concurrency-related vulnerabilities is generated by defects in the sharing of resources such as memory, devices, or files. In such a case, the defect may be a design error associated with the concurrency control mechanisms or with an implementation error, such as not correctly using those mechanisms. Caching errors can be considered members of this class.

Strictly speaking, signal-handling defects are not concurrency defects. Signal handlers are invoked preemptively in the main thread of the process, so signal handlers are not really concurrently executed. However, from the developer's viewpoint, they "feel" like concurrent execution, so we classify them here, at least for now.

Applicable Context

All of the following must be true:

- A "system" must have multiple concurrently operating threads of control.
- Two or more of those threads must use a shared data object, device, or other resource.
- At least one thread must use the shared resource without using the appropriate concurrency control mechanism correctly (or at all).

| ***Impact*** | ***Minimally*** | ***Maximally*** |
|---|---|---|
| 1 | None | Deadlock: One or more threads may become permanently blocked. |
| 2 | None | Loss of information: Saved information is overwritten by another thread. |
| 3 | None | Loss of integrity of information: Information written by multiple threads may be arbitrarily interlaced. |
| 4 | None | Loss of liveness: Imbalance in access to shared resources by competing threads can cause performance problems. |

#### Security Policies to Be Preserved

1. Threads must not deadlock.
2. Information must not be lost.
3. Information must not be corrupted.
4. Acceptable performance must be maintained.

#### How to Recognize This Defect

Concurrency defects are extremely difficult to recognize. There is no general-purpose approach to finding them.

| ***Efficacy*** | ***Mitigation*** |
|---|---|
| Low | Where no concurrency control mechanism is available, seek to minimize the interval between the time of check and the time of use. Technically this action does not correct the problem, but it can make the error much more difficult to exploit. |
| Infinite | The appropriate concurrency control mechanism must be used in the *conventional* way (assuming there is one). |

### Recapping Security Guidelines

Security guidelines represent prescriptive guidance for those software design-level concerns typically faced by the software development project team. They provide an effective complement to the more abstract and far-reaching security principles. When combined, these two resources provide the knowledge necessary to know what you *should* do when creating software architecture and design. When combined with representations of the attacker's perspective such as attack patterns (described in Section 4.3.3), they also can provide an effective checklist of what you may have failed to do to ensure that your software is both resistant and resilient to likely attack.

# Attack Patterns

**Attack patterns** are another knowledge resource available to the software project manager. They offer a formalized mechanism for representing and communicating the attacker's perspective by describing approaches commonly taken to attack software. This attacker's perspective, while important throughout the SDLC, has increased value during architecture and design phase. Attack patterns offer a valuable resource during three primary activities of architecture and design: design and security pattern selection, threat modeling, and attack resistance.

One of the key methods for improving the stability, performance, and security of a software architecture is the leveraging of proven patterns. Appropriate selection of design patterns and security patterns can offer significant architectural risk mitigation. Comprehensive attack patterns can facilitate this process through identification of prescribed and proscribed design and security patterns known to mitigate the risk of given attack patterns that have been determined to be applicable to the software under development.

During the asset identification and threat modeling (combining software characterization and threat analysis) portions of architectural risk analysis, the architects/analysts identify and characterize the assets of the software, the relevant potential threats they may face, and the topology and nature of the software's attack surface and trust boundaries. They combine these considerations into an integrated model that demonstrates likely vectors of attack against the attack surface by the identified threats and targeted at the identified assets. These attack vectors represent security risks. One of the more effective mechanisms for quickly identifying likely attack vectors, characterizing their nature, and identifying vetted mitigation approaches is the use of attack patterns as an integral part of the threat model.

Lastly, during the attack resistance analysis portion of the risk analysis process, in which the architectural security is vetted, attack patterns can be a valuable tool in identifying and characterizing contextually appropriate attacker perspectives to consider in a red teaming type of approach.

## Summary

The architecture and design phase of the SDLC represents a critical time for identifying and preventing security flaws before they become part of the software. As the connectivity, complexity, and extensibility of software increase, the importance of effectively addressing security concerns as an integral part of the architecture and design process will become even more critical. During this phase in the software development effort, architects, designers, and security analysts have an opportunity to ensure that requirements are interpreted appropriately through a security lens and that appropriate security knowledge is leveraged to give the software structure and form in a way that minimizes security risk. This can be accomplished through use of the following practices and knowledge resources:

- Beyond consistent application of good security common sense, one of the greatest opportunities for reducing the security risk of software early in the SDLC is through the practice of *architectural risk analysis*. By carrying out software characterization, threat analysis, architectural vulnerability assessment, risk likelihood determination, risk impact determination, and risk mitigation planning, project team members can identify, prioritize, and implement appropriate mitigating controls, revisions, or actions before security problems take root. These activities and their by-products not only yield more secure software, but also provide the software development team with a much richer understanding of how the software is expected to behave under various operating conditions and how its architecture and design support that behavior.
- The activities and processes that make up architectural risk analysis are enabled and fueled by a range of experiential security knowledge without which they would be meaningless. **Security principles** and **security guidelines** offer prescriptive guidance that serves as a positive benchmark against which to compare and assess software's architecture and design. The high-level abstraction, universal context, and lasting nature of security principles make the wisdom they bring to architectural-level decisions extremely valuable. The broad brush of security principles is enhanced and refined through the consideration of context-specific security guidelines that provide more concrete guidance down into the design level.
- Architectural risk analysis requires not only recognition of good defensive software practices, but also a solid understanding of the sort of threats and attacks that the software is likely to face. *Attack patterns*, describing common approaches to attacking software, provide one of the most effective resources for capturing this attacker's perspective. By developing the software's architecture and design to be resistant and resilient to the types of attack it is likely to face, the software development team lays a solid security foundation on which to build.