Chapter 2. What Makes Software Secure?

E M L

2.1. Introduction

To answer the question, "What makes software secure?" it is important to understand the meaning of software security in the broader context of software assurance.

As described in **Chapter 1**, software assurance is the domain of working toward software that exhibits the following qualities:

• Trustworthiness, whereby no exploitable vulnerabilities or weaknesses exist, either of malicious or unintentional origin

• Predictable execution, whereby there is justifiable confidence that the software, when executed, functions as intended

• Conformance, whereby a planned and systematic set of multidisciplinary activities ensure that software processes and products conform to their requirements, standards, and procedures

We will focus primarily on the dimension of trustworthiness—that is, which properties can be identified, influenced, and asserted to characterize the trustworthiness, and thereby the security, of software. To be effective, predictable execution must be interpreted with an appropriately broader brush than is typically applied. Predictable execution must imply not only that software effectively does what it is expected to do, but also that it is robust under attack and does not do anything that it is *not* expected to do. This may seem to some to be splitting hairs, but it is an important distinction between what makes for high-quality software versus what makes for secure software.

To determine and influence the trustworthiness of software, it is necessary to define the properties that characterize secure software, identify mechanisms to influence these properties, and leverage structures and tools for asserting the presence or absence of these properties in communication surrounding the security of software.

This chapter draws on a diverse set of existing knowledge to present solutions to these challenges and provide you with resources to explore for more in-depth coverage of individual topics.

E M L L4

2.2. Defining Properties of Secure Software[1]

Before we can determine the security characteristics of software and look for ways to effectively measure and improve them, we must first define the properties by which these characteristics can be described. These properties consist of (1) a set of core properties whose presence (or absence) are the ground truth that makes software secure (or not) and (2) a set of influential properties that do not directly make software secure but do make it possible to characterize how secure software is.

2.2.1. Core Properties of Secure Software

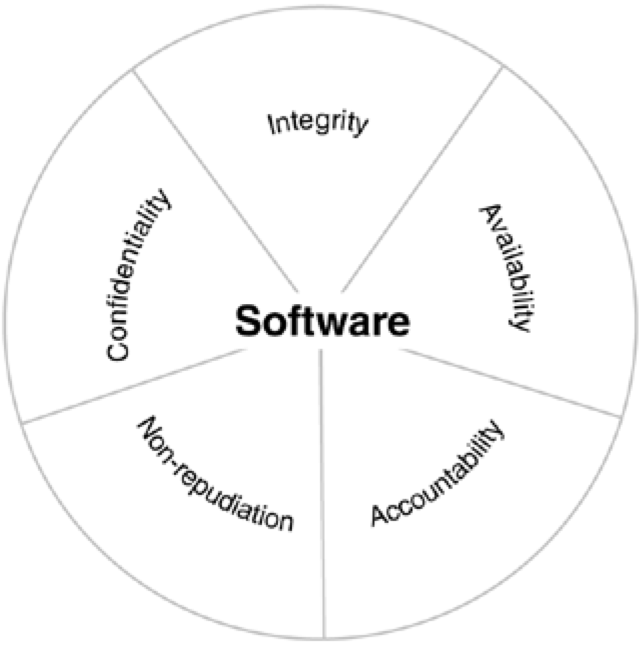
Several fundamental properties may be seen as attributes of security as a software property, as shown in Figure 2–1:

- **Confidentiality**. The software must ensure that any of its characteristics (including its relationships with its execution environment and its users), its managed assets, and/or its content are obscured or hidden from *unauthorized* entities. This remains appropriate for cases such as open-source software; its characteristics and content are available to the public (authorized entities in this case), yet it still must maintain confidentiality of its managed assets.

- **Integrity**. The software and its managed assets must be resistant and resilient to subversion. Subversion is achieved through unauthorized modifications to the software code, managed assets, configuration, or behavior by authorized entities, or any modifications by unauthorized entities. Such modifications may include overwriting, corruption, tampering, destruction, insertion of unintended (including malicious) logic, or deletion. Integrity must be preserved both during the software's development and during its execution.

• **Availability**. The software must be operational and accessible to its intended, authorized users (humans and processes) whenever it is needed. At the same time, its functionality and privileges must be inaccessible to unauthorized users (humans and processes) at all times.

Figure 2-1. *Core security properties of secure software*



Two additional properties commonly associated with human users are required in software entities that act as users (e.g., proxy agents, Web services, peer processes):

• **Accountability**. All security-relevant actions of the software-as-user must be recorded and tracked, with attribution of responsibility. This tracking must be possible both while and after the recorded actions occur. The audit-related language in the security policy for the software system should indicate which actions are considered "security relevant."

• **Non-repudiation**. This property pertains to the ability to prevent the software-as-user from disproving or denying responsibility for actions it has performed. It ensures that the accountability property cannot be subverted or circumvented.

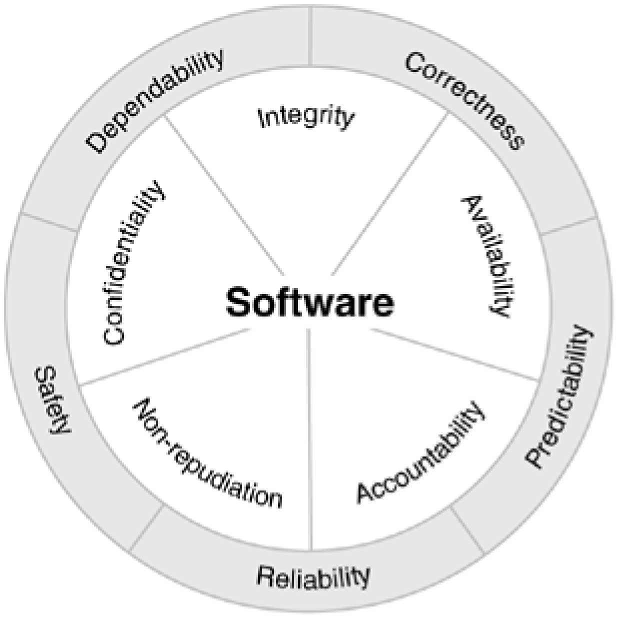
These core properties are most typically used to describe network security. However, their definitions have been modified here slightly to map these still valid concepts to the software security domain. The effects of a security breach in software can, therefore, be described in terms of the effects on these core properties. A successful SQL injection attack on an application to extract personally identifiable information from its database would be a violation of its confidentiality property. A successful cross-site scripting (XSS) attack against a Web application could result in a violation of both its integrity and availability properties. And a successful buffer overflow[2] attack that injects malicious code in an attempt to steal user account information and then alter logs to cover its tracks would be a violation of all five core security properties. While many other important characteristics of software have implications for its security, their relevance can typically be described and communicated in terms of how they affect these core properties.

2.2.2. Influential Properties of Secure Software

Some properties of software, although they do not directly make software secure, nevertheless make it possible to characterize how secure software is (Figure 2–2):

- Dependability
- Correctness
- Predictability
- Reliability
- Safety

Figure 2-2. *Influential properties of secure software*



These influential properties are further influenced by the size, complexity, and traceability of the software. Much of the activity of software security engineering focuses on addressing these properties and thus targets the core security properties themselves.

Dependability and Security

In simplest terms, dependability is the property of software that ensures that the software always operates as intended. It is not surprising that security as a property of software and dependability as a property of software share a number of subordinate properties (or attributes). The most obvious, to security practitioners, are availability and integrity. However, according to Algirdas Avizienis et al. in “Basic Concepts and Taxonomy of Dependable and Secure Computing,” a number of other properties are shared by dependability and security, including reliability, safety, survivability, maintainability, and fault tolerance **[Avizienis 2004]**.

To better understand the relationship between security and dependability, consider the nature of risk to security and, by extension, dependability. A variety of factors affect the defects and weaknesses that lead to increased risk related to the security or dependability of software. But are they human-made or environmental? Are they intentional or unintentional? If they are intentional, are they malicious? Nonmalicious intentional weaknesses often result from bad judgment. For example, a software engineer may make a tradeoff between performance and usability on the one hand and security on the other hand that results in a design decision that includes weaknesses. While many defects and weaknesses have the ability to affect both the security and the dependability of software, it is typically the intentionality, the exploitability, and the resultant impact if exploited that determine whether a defect or weakness actually constitutes a vulnerability leading to security risk.

Note that while dependability directly implies the core properties of integrity and availability, it does not necessarily imply confidentiality, accountability, or non-repudiation.

Correctness and Security

From the standpoint of quality, correctness is a critical attribute of software that should be consistently demonstrated under all anticipated operating conditions. Security requires that the attribute of correctness be maintained under unanticipated conditions as well. One of the mechanisms most commonly used to attack the security of software seeks to cause the software's correctness to be violated by forcing it into unanticipated operating conditions, often through unexpected input or exploitation of environmental assumptions.

Some advocates for secure software engineering have suggested that good software engineering is all that is needed to ensure that the software produced will be free of exploitable faults and other weaknesses. There is a flaw in this thinking—namely, good software engineering typically fails to proactively consider the behavior of the software under unanticipated conditions. These unanticipated conditions are typically determined to be out of scope as part of the requirements process. Correctness under anticipated conditions (as it is typically interpreted) is not enough to ensure that the software is secure, because the conditions that surround the software when it comes under attack are very likely to be unanticipated. Most software specifications do not include explicit requirements for the software's functions to continue operating correctly under unanticipated conditions. Software engineering that focuses only on achieving correctness under anticipated conditions, therefore, does not ensure that the software will remain correct under unanticipated conditions.

If explicit requirements for secure behavior are not specified, then requirements-driven engineering, which is used frequently to increase the correctness of software, will do nothing to ensure that correct software is also secure. In requirements-driven engineering, correctness is assured by verifying that the software operates in strict accordance with its specified requirements. If the requirements are deficient, the software still may strictly be deemed correct as long as it satisfies those requirements that do exist.

The requirements specified for the majority of software are limited to functional, interoperability, and performance requirements. Determining that such requirements have been satisfied will do nothing to ensure that the software will also behave securely even when it operates correctly. Unless a requirement exists for the software to contain a particular security property or attribute, verifying correctness will indicate nothing about security. A property or attribute that is not captured as a requirement will not be the subject of any verification effort that seeks to discover whether the software contains that function or property.

Security requirements that define software's expected behavior as adhering to a desired security property are best elicited through a documented process, such as the use of misuse/abuse cases (see **Section 3.2**). Misuse/abuse cases are descriptive statements of the undesired, nonstandard conditions that the software is likely to face during its operation from either unintentional misuse or intentional and malicious misuse/abuse. Misuse/abuse cases are effectively captured by analyzing common approaches to attack that the software is likely to face. Attack patterns, as discussed later in this chapter, are a physical representation of these common approaches to attack. Misuse/abuse cases, when explicitly captured as part of the requirements process, provide a measurable benchmark against which to assess the completeness and quality of the defined security requirements to achieve the desired security properties in the face of attack and misuse.

It is much easier to specify and satisfy functional requirements stated in positive terms ("The software will perform such-and-such a function"). Security properties and attributes, however, are often nonfunctional

("This process must be non-bypassable"). Even "positively" stated requirements may reflect inherently negative concerns. For example, the requirement "If the software cannot handle a fault, the software must release all of its resources and then terminate execution" is, in fact, just a more positive way of stating the requirement that "A crash must not leave the software in an insecure state."

Moreover, it is possible to specify requirements for functions, interactions, and performance attributes that result in insecure software behavior. By the same token, it is possible to implement software that deviates from its functional, interoperability, and performance requirements (that is, software that is incorrect only from a requirements engineering perspective) without that software actually behaving insecurely.

Software that executes correctly under anticipated conditions cannot be considered secure when it is used in an operating environment characterized by unanticipated conditions that lead to unpredictable behavior. However, it may be possible to consider software that is *in* correct but completely predictable to be secure *if* the incorrect portions of the software are not manifested as vulnerabilities. Thus it does not follow that correctness will necessarily help assure security, or that incorrectness will necessarily become manifest as insecurity. Nevertheless, correctness in software is just as important a property as security. Neither property should ever have to be achieved at the expense of the other.

A number of vulnerabilities in software that can be exploited by attackers can be avoided by engineering for correctness. By reducing the total number of defects in software, the subset of those defects that are exploitable (that is, are vulnerabilities) will be coincidentally reduced. However, some complex vulnerabilities may result from a sequence or combination of interactions among individual components; each interaction may be perfectly correct yet, when combined with other interactions, may result in incorrectness and vulnerability. Engineering for correctness will not eliminate such complex vulnerabilities.

For the purposes of requirements-driven engineering, no requirement for a software function, interface, performance attribute, or any other attribute of the software should ever be deemed "correct" if that requirement can only be satisfied in a way that allows the software to behave insecurely or that makes it impossible to determine or predict whether the software will behave securely. Instead, every requirement should be specified in a way that ensures that the software will always and only behave securely when the requirement is satisfied.

“Small” Faults, Big Consequences

There is a conventional wisdom espoused by many software engineers that says vulnerabilities which fall within a specified range of speculated impact (“size”) can be tolerated and allowed to remain in the software. This belief is based on the underlying assumption that small faults have small consequences. In terms of defects with security implications, however, this conventional wisdom is wrong. Nancy Leveson suggests that vulnerabilities in large software-intensive systems with significant human interaction will increasingly result from multiple minor defects, each insignificant by itself, thereby collectively placing the system into a vulnerable state **[Leveson 2004]**.

Consider a classic stack-smashing attack that relies on a combination of multiple “small” defects that individually may have only minor impact, yet together represent significant vulnerability **[Aleph One 1996]**. An input function writes data to a buffer without first performing a bounds check on the data. This action occurs in a program that runs with root privilege. If an attacker submits a very long string of input data that includes both malicious code and a return address pointer to that code, because the program does not do bounds checking, the input will be accepted by the program and will overflow the stack buffer that receives it. This outcome will allow the malicious code to be loaded onto the program’s execution stack and overwrite the subroutine return address so that it points to that malicious code. When the subroutine terminates, the program will jump to the malicious code, which will be executed, operating with root privilege. This particular malicious code is written to call the system shell, enabling the attacker to take control of the system. (Even if the original program had not operated with root privileges, the malicious code may have contained a privilege escalation exploit to gain those privileges.)

Obviously, when considering software security, the *perceived size* of a vulnerability is not a reliable predictor of the magnitude of that vulnerability *impact*. For this reason, the risks of every known vulnerability—regardless of whether it is detected during design review, implementation, or testing—should be explicitly analyzed and mitigated or accepted by authoritative persons in the development organization. Assumption is a primary root of insecurity anywhere, but especially so in software.

For high-assurance systems, there is no justification for tolerating known vulnerabilities. True software security is achievable only when all known aspects of the software are understood and verified to be predictably correct. This includes verifying the correctness of the software's behavior under a wide variety of conditions, including hostile conditions. As a consequence, software testing needs to include observing the software's behavior under the following circumstances:

• Attacks are launched against the software itself

• The software's inputs or outputs (e.g., data files, arguments, signals) are compromised

• The software's interfaces to other entities are compromised

• The software's execution environment is attacked

Predictability and Security

Predictability means that the software's functionality, properties, and behaviors will always be what they are expected to be as long as the conditions under which the software operates (i.e., its environment, the inputs it receives) are also predictable. For dependable software, this means the software will never deviate from correct operation under anticipated conditions.

Software security extends predictability to the software's operation under unanticipated conditions—specifically, under conditions in which attackers attempt to exploit faults in the software or its environment. In such circumstances, it is important to have confidence in precisely how the software will behave when faced with misuse or attack. The best way to ensure predictability of software under unanticipated conditions is to minimize the presence of vulnerabilities and other weaknesses, to prevent the insertion of malicious logic, and to isolate the software to the greatest extent possible from unanticipated environmental conditions.

Reliability, Safety, and Security[3]

The focus of reliability for software is on preserving predictable, correct execution despite the presence of unintentional defects and other weaknesses and unpredictable environment state changes. Software that is highly reliable is often referred to as high-confidence software (implying that a high level of assurance of that reliability exists) or fault-tolerant software (implying that fault tolerance techniques were used to achieve the high level of reliability).

Software safety depends on reliability and typically has very real and significant implications if the property is not met. The consequences, if reliability is not preserved in a safety-critical system, can be catastrophic: Human life may be lost, or the sustainability of the environment may be compromised.

Software security extends the requirements of reliability and safety to the need to preserve predictable, correct execution even in the face of *malicious* attacks on defects or weaknesses and environmental state changes. It is this *maliciousness* that makes the requirements of software security somewhat different from the requirements of safety and reliability. Failures in a reliability or safety context are expected to be random and unpredictable. Failures in a security context, by contrast, result from human effort (direct, or through malicious code). Attackers tend to be persistent, and once they successfully exploit a vulnerability, they tend to continue exploiting that vulnerability on other systems as long as the vulnerability is present and the outcome of the attack remains satisfactory.

Until recently, many software reliability and safety practitioners have not concerned themselves with software security issues. Indeed, the two domains have traditionally been viewed as separate and distinct. The truth is that safety, as a property of software, is directly dependent on security properties such as dependability. A failure in the security of software, especially one that is intentional and malicious, can directly change the operational and environmental presumptions on which safety is based, thereby compromising any possible assurance in its safety properties. Any work toward assuring the safety of software that does not take security properties into consideration is incomplete and unreliable.

Size, Complexity, Traceability, and Security

Software that satisfies its requirements through simple functions that are implemented in the smallest amount of code that is practical, with process flows and data flows that are easily followed, will be easier to comprehend and maintain. The fewer the dependencies in the software, the easier it will be to implement effective failure detection and to reduce the **attack surface**.[4]

Size and complexity should be not only properties of the software's implementation, but also properties of its design, as they will make it easier for reviewers to discover design flaws that could be manifested as exploitable weaknesses in the implementation. Traceability will enable the same reviewers to ensure that the design satisfies the specified security requirements and that the implementation does not deviate from the secure design. Moreover, traceability provides a firm basis on which to define security test cases.

M L L3

2.3. How to Influence the Security Properties of Software

Once you understand the properties that determine the security of software, the challenge becomes acting effectively to influence those properties in a positive way. The ability of a software development team to manipulate the security properties of software resolves to a balance between engaging in defensive action and thinking like an attacker. The primary perspective is that of a defender, where the team works to build into the software appropriate security features and characteristics to make the software more resistant to attack and to minimize the inherent weaknesses in the software that may make it more vulnerable to attack. The balancing perspective is that of the attacker, where the team strives to understand the exact nature of the threat that the software is likely to face so as to focus defensive efforts on areas of highest risk. These two perspectives, working in combination, guide the actions taken to make software more secure.

Taking action to address these perspectives requires knowledge resources (prescriptive, diagnostic, and historical) covering the various aspects of software assurance combined with security best practices called **touch-points** integrated throughout the SDLC; all of this must then be deployed under an umbrella of applied risk management **[McGraw 2006]**. For the discussion here, we use these definitions for best practices and touchpoints:

> Best practices are the most efficient (least amount of effort) and effective (best results) way of accomplishing a task, based on repeatable procedures that have proven themselves over time for large numbers of people.[5]
> Touchpoints are lightweight software security best practice activities that are applied to various software artifacts. **[McGraw 2006]**

Project managers who are concerned with the security of the software they are developing must proactively select the appropriate practices and knowledge to ensure that both the defensive and attacker's perspectives are appropriately represented and understood by the development team. This chapter, and the rest of this book, presents several of the most effective options (though they vary in their level of adoption) for knowledge resources and security practices and guidance on how to select and use them.

2.3.1. The Defensive Perspective

Assuming the defensive perspective involves looking at the software from the inside out. It requires analyzing the software for vulnerabilities and opportunities for the security of the software to be compromised through inadvertent misuse and, more importantly, through malicious attack and abuse. Doing so requires the software development team to perform the following steps:

- Address expected issues through the application of appropriate security architecture and features

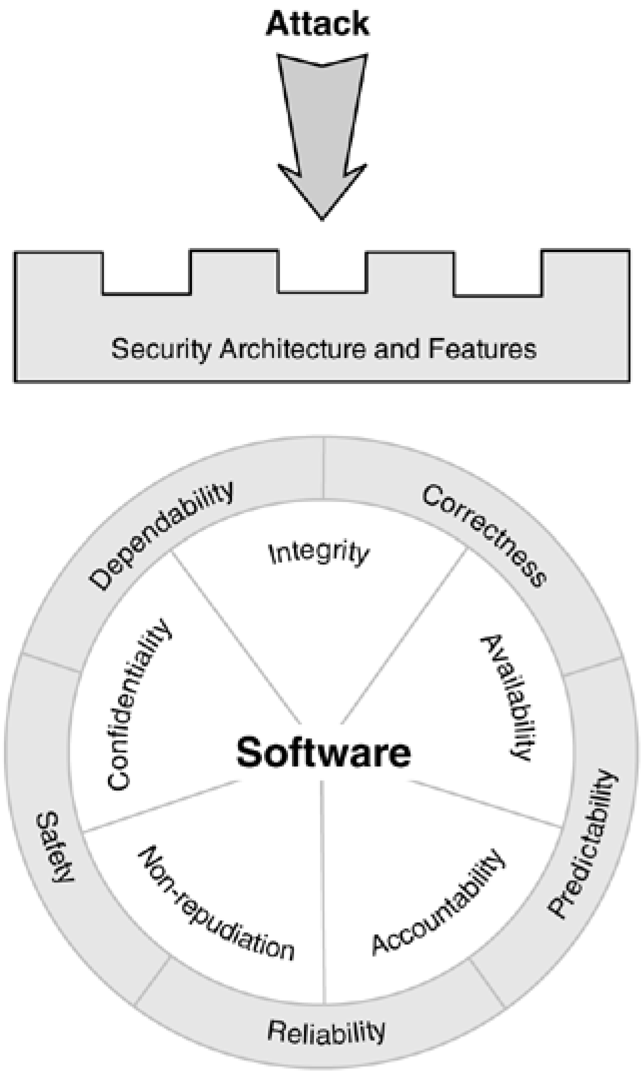
- Address unexpected issues through the avoidance, removal, and mitigation of weaknesses that could lead to security vulnerabilities

- Continually strive to improve and strengthen the attack resistance, tolerance, and resilience of the software in everything they do

Addressing the Expected: Security Architecture and Features

When most people think of making software secure, they think in terms of the architecture and functionality of security features. Security features and functionality alone are insufficient to ensure software security, but they are a necessary facet to consider. As shown in Figure 2–3, security features aim to address expected security issues with software such as authentication, authorization, access control, permissions, privileges, and cryptography. Security architecture is the overall framework that holds these security functionalities together and provides the set of interfaces that integrates them with the broader software architecture.[6]

Figure 2-3. *Addressing expected issues with security architecture and features*



Without security architecture and features, adequate levels of confidentiality, integrity, accountability, and non-repudiation may be unattainable. However, fully addressing these properties (as well as availability) requires the development team not only to provide functionality to manage the security behavior of the software, but also to ensure that the functionality and architecture of the software do not contain weaknesses that could render the software vulnerable to attack in potentially unexpected ways.

Addressing the Unexpected: Avoiding, Removing, and Mitigating Weaknesses

Many activities and practices are available across the life cycle of software systems that can help reduce and mitigate weaknesses present in software. These activities and practices can typically be categorized into two approaches: application defense and software security.

Application Defense

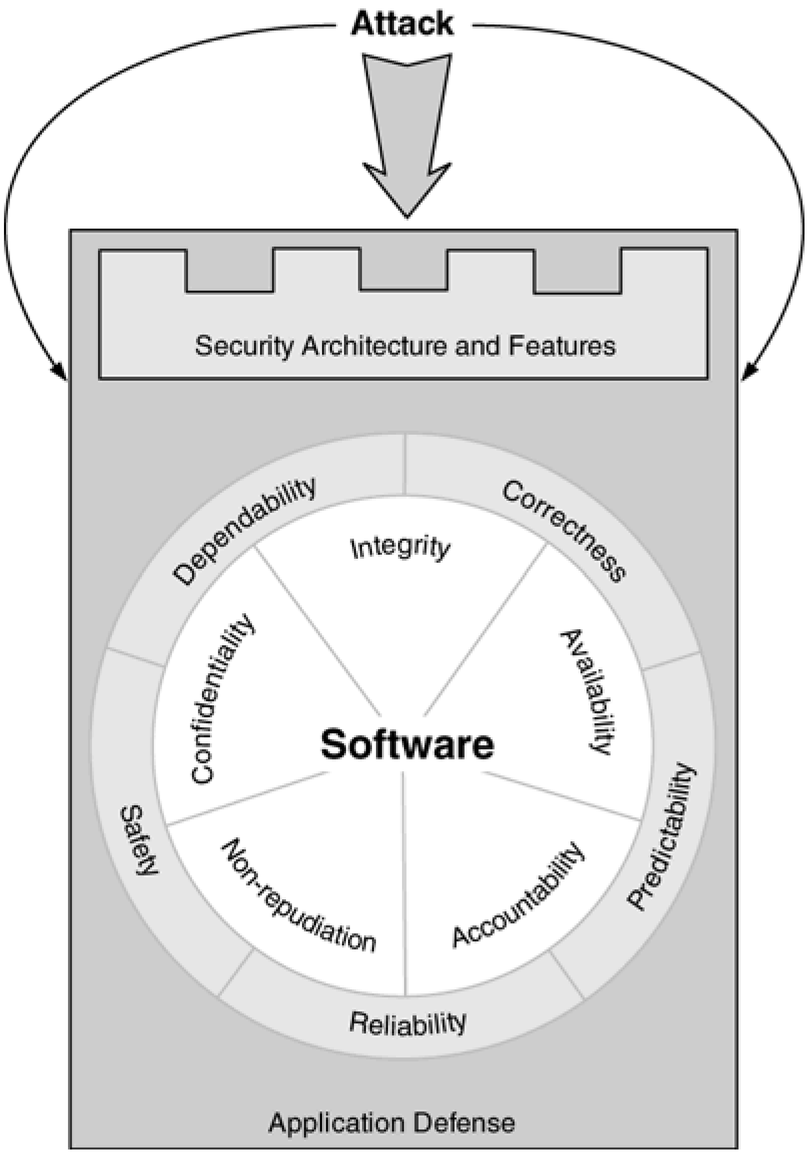
Employing practices focused at detecting and mitigating weaknesses in software systems after they are deployed is often referred to as *application defense* (see **Figure 2–4**), which in many cases is mislabeled as *application security*. Application defense techniques typically focus on the following issues:

• Establishing a protective boundary around the application that enforces rules defining valid input or recognizes and either blocks or filters input that contains recognized patterns of attack

• Constraining the extent and impact of damage that might result from the exploit of a vulnerability in the application

• Discovering points of vulnerability in the implemented application through black-box testing so as to help developers and administrators identify necessary countermeasures (see the description of black-box testing in **Section 5.5.4**)

Figure 2-4. *Addressing the unexpected through application defense*



Reactive application defense measures are often similar to the techniques and tools used for securing networks, operating systems, and middleware services. They include things such as vulnerability scanners, intrusion detection tools, and firewalls or security gateways. Often, these measures are intended to strengthen the boundaries *around* the application rather than address the actual vulnerabilities *inside* the application.

In some cases, application defense measures are applied as stopgaps for from-scratch application software until a security patch or new version is released. In other cases, these measures provide ongoing defense in depth to counter vulnerabilities in the application. In software systems that include acquired or reused (commercial, government off-the-shelf, open-source, shareware, freeware, or legacy) binary components, application defense techniques and tools may be the only cost-effective countermeasures to mitigate vulnerabilities in those components.

Application defense as typically practiced today incorporates few, if any, techniques and tools that will aid the developer in producing software that has very few vulnerabilities in the first place. These practices often provide valuable guidance in identifying and mitigating more obvious security vulnerabilities, especially those associated with the deployment configuration and environment. Most serious weaknesses, including both design flaws and implementation bugs, are not typically detectable in this manner, however; when they are, they are usually much more expensive to remedy so late in the life cycle. A software security perspective, by contrast, not only incorporates protective, post-implementation techniques, but also addresses the need to specify, design, and implement an application with a minimal attack surface.

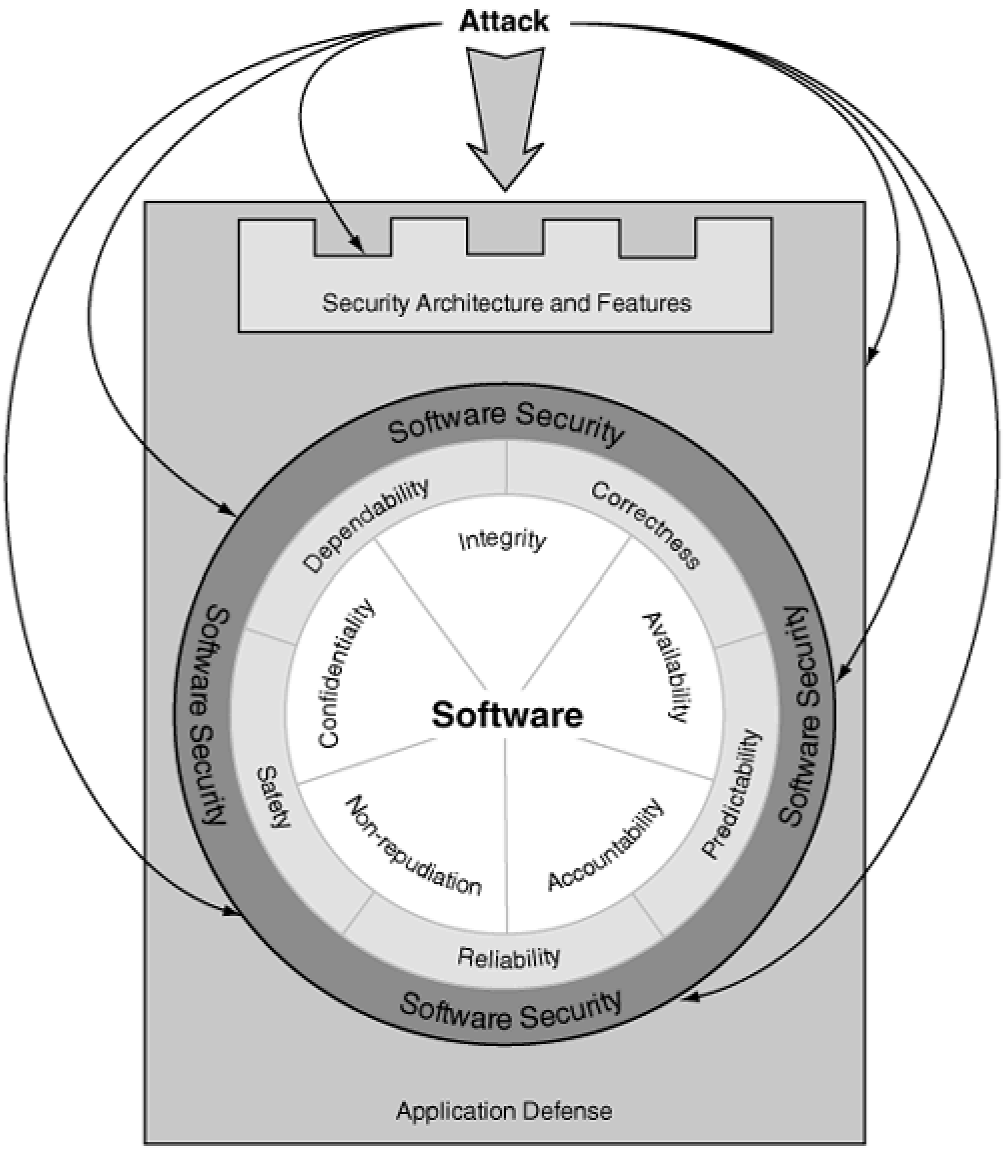
The point to take away is that a disciplined, repeatable, security-enhanced development process should be instituted that ensures application defense measures are used only because they are determined in the design process to be the best approach to solving a software security problem, not because they are the only possible approach after the software is deployed.

That said, using secure systems engineering approaches can be helpful to further protect securely engineered software in deployment by reducing its exposure to threats in various operational environments. These measures may be particularly useful for reducing risk for software, such as commercial and open-source software, that is intended to be deployed in a wide variety of threat environments and operational contexts.

Software Security

While application defense takes a somewhat after-the-fact approach, practices associated with “software security” and its role in secure software engineering processes focus on preventing weaknesses from entering the software in the first place or, if that is unavoidable, at least removing them as early in the life cycle as possible and before the software is deployed (see Figure 2–5). These weaknesses, whether unintentional or maliciously inserted, can enter the software at any point in the development process through inadequate or incorrect requirements; ambiguous, incomplete, unstable, or improper architecture and design; implementation errors; incomplete or inappropriate testing; or insecure configuration and deployment decisions.

Figure 2-5. *Addressing the unexpected through software security*



Contrary to a common misconception, software security cannot be the sole responsibility of the developers who are writing code, but rather requires the involvement of the entire development team and the organization supporting it. Luckily, a wide variety of security-focused practices are available to software project managers and their development teams that can be seamlessly integrated throughout any typical software engineering SDLC. Among other things, these practices include security requirements engineering with misuse/abuse cases, architectural risk analysis, secure code review, risk-based security testing, and software penetration testing. These practices of software security, which are collectively referred to as “building security in,” are the primary focus of this book. The chapters that follow outline software security practices and knowledge associated with various phases of the SDLC that are of value to development teams looking to minimize weaknesses and thereby build more secure software that is resistant, tolerant, and resilient to attack.

Attack Resistance, Attack Tolerance, and Attack Resilience

The ultimate goal of defensive software security efforts can be most clearly seen in their ability to maintain security properties in the face of motivated and intentional attempts to subvert them. The ability of software to function in the face of attack can be broken down into three primary characteristics: attack resistance, attack tolerance, and attack resilience.

• **Attack resistance** is the ability of the software to prevent the capability of an attacker to execute an attack against it. The most critical of the three characteristics, it is nevertheless often the most difficult to achieve, as it involves minimizing exploitable weaknesses at all levels of abstraction, from architecture through detailed implementation and deployment. Indeed, sometimes attack resistance is impossible to fully achieve.

• **Attack tolerance** is the ability of the software to “tolerate” the errors and failure that result from successful attacks and, in effect, to continue to operate as if the attacks had not occurred.

• **Attack resilience** is the ability of the software to isolate, contain, and limit the damage resulting from any failures caused by attack-triggered faults that the software was unable to resist or tolerate and to recover as quickly as possible from those failures.[7]

Attack tolerance and attack resilience are often a result of effective architectural and design decisions rather than implementation wizardry. Software that can achieve attack resistance, attack tolerance, and attack resilience is implicitly more capable of maintaining its core security properties.

2.3.2. The Attacker’s Perspective[8]

Assuming the attacker's perspective involves looking at the software from the outside in. It requires thinking like attackers think, and analyzing and understanding the software the way they would to attack it. Through better understanding of how the software is likely to be attacked, the software development team can better harden and secure it against attack.

The Attacker's Advantage

The primary challenge in building secure software is that it is much easier to find vulnerabilities in software than it is to make software secure. As an analogy, consider a bank vault. Its designers need to ensure that it is safe against many different types of attacks, not just the seemingly obvious ones. It must generally be safe against mechanical attacks (e.g., using bulldozers), explosives, and safecracking, to name a few, while still maintaining usability (e.g., allowing authorized personnel to enter, having sufficient ventilation and lighting). This is clearly not a trivial task. However, the attacker may simply need to find one exploitable vulnerability to achieve his or her goal of entering the vault. The attacker may try to access the vault through various potential means, including through the main entrance by cracking the safe combination, through the ceiling, by digging underground, by entering through the ventilation system, by bribing an authorized employee to open the vault, or by creating a small fire in the bank while the vault is open to cause all employees to flee in panic. Given these realities, it is evident that building and maintaining bank vault security is typically much more difficult than breaking into a vault.

Building secure software has similar issues, but the problem is exacerbated by the virtual (rather than physical) nature of software. With many systems, the attacker may actually possess the software (obtaining a local copy to attack is often trivial) or could attack it from anywhere in the world through networks. Given attackers' ability to attack remotely and without physical access, vulnerabilities become much more widely exposed to attack. Audit trails may not be sufficient to catch attackers after an attack takes place, because attackers could leverage the anonymity of an unsuspecting user's wireless network or public computers to launch attacks.

The attackers' advantage is further strengthened by the fact that attackers have been learning how to exploit software for several decades, but the general software development community has not kept up-to-date with the knowledge that attackers have gained. This knowledge gap is also evident in the difference of perspective evident between attackers, with their cynical deconstructive view, and developers, with their happy-go-lucky "You're not supposed to do that" view. The problem continues to grow in part because of the traditional fear that teaching how software is exploited could actually reduce the security of software by helping the existing attackers and even potentially creating new ones. In the past, the software development community hoped that obscurity would keep the number of attackers relatively small. This assumption has been shown to be a poor one, and some elements of the community are now beginning to look for more effective methods of addressing this problem.

To identify and mitigate vulnerabilities in software, the development community needs more than just good software engineering and analytical practices, a solid grasp of software security features, and a powerful set of tools. All of these things are necessary but not sufficient. To be effective, the community needs to think creatively and to have a firm grasp of the attacker's perspective and the approaches used to exploit software [Hoglund 2004; Koizol 2004].

Finding a Way to Represent the Attacker's Perspective

For software development teams to take advantage of the attacker's perspective in building security into software, there first must be a mechanism for capturing and communicating this perspective from knowledgeable experts and communicating it to teams. A powerful resource for providing such a mechanism is the attack pattern.

Design patterns are a familiar tool used by the software development community to help solve recurring problems encountered during software development [Alexander 1964, 1977, 1979; Gamma 1995]. These patterns attempt to tackle head-on the thorny problems of secure, stable, and effective software architecture and design. Since the introduction of design patterns, the pattern construct has been applied to many other areas of software development. One of these areas is software security and representation of the attacker's perspective in the form of attack patterns. The term **attack patterns** was coined in discussions among software security experts starting around 2001, was introduced in the paper *Attack Modeling for Information Security and Survivability* **[Moore 2001]**, and was brought to the broader industry in greater detail and with a solid set of specific examples by Greg Hoglund and Gary McGraw in their book *Exploiting Software: How to Break Code* **[Hoglund 2004]**.

Attack patterns apply the problem–solution paradigm of design patterns in a destructive—rather than constructive—context. Here, the common problem targeted by the pattern represents the objective of the software attacker, and the pattern's solution represents common methods for performing the attack. In short, attack patterns describe the techniques that attackers might use to break software.

The incentive behind using attack patterns is that they give software developers a structured representation of how attackers think, which enables them to anticipate attacks and hence take more effective steps to mitigate the likelihood or impact of attacks. Attack patterns help to categorize attacks in a meaningful way so that problems and solutions can be discussed effectively. They can identify the types of known attacks to which an application could be exposed so that mitigations can be built into the application. Another benefit of attack patterns is that they contain sufficient detail about how attacks are carried out to enable developers to help prevent them. Owing to the omission of information about software security in many curricula and the traditional shroud of secrecy surrounding exploits, software developers are often ill informed about the field of software security and especially software exploit. The concept of attack patterns can be used to teach the software development community both how software is exploited in reality and how to avoid such attacks.

Since the publication of *Exploiting Software*, several individuals and groups in the industry have tried to push the concept of attack patterns forward, with varying levels of success. These efforts have faced challenges such as the lack of a common definition and schema for attack patterns, a lack of diversity in the targeted areas of analysis by the various groups involved, and a lack of any independent body to act as the collector and disseminator of common attack pattern catalogues. The two most significant advances in this regard have been the recent publication of the detailed attack pattern articles on the Build Security In Web site sponsored by the U.S. Department of Homeland Security (DHS) and the initial launch of the ongoing DHS-sponsored Common Attack Pattern Enumeration and Classification (CAPEC) **[CAPEC 2007]** initiative content. Content released as part of the initial launch of CAPEC includes a formal attack pattern schema, a draft attack classification taxonomy, and 101 actual detailed attack patterns. All of this content is freely available to the public to use for software security engineering.

MITRE Security Initiatives

In addition to the Common Attack Pattern Enumeration and Classification (CAPEC), the Making Security Measurable program sponsored by the Department of Homeland Security and led by MITRE Corporation produces *Common Vulnerabilities and Exposures* (CVE), a dictionary of publicly known information security vulnerabilities and exposures, and *Common Weakness Enumeration* (CWE), a dictionary of software weakness types. Links to all three can be found on MITRE's Making Security Measurable site, along with links to other information security enumerations, languages, and repositories.

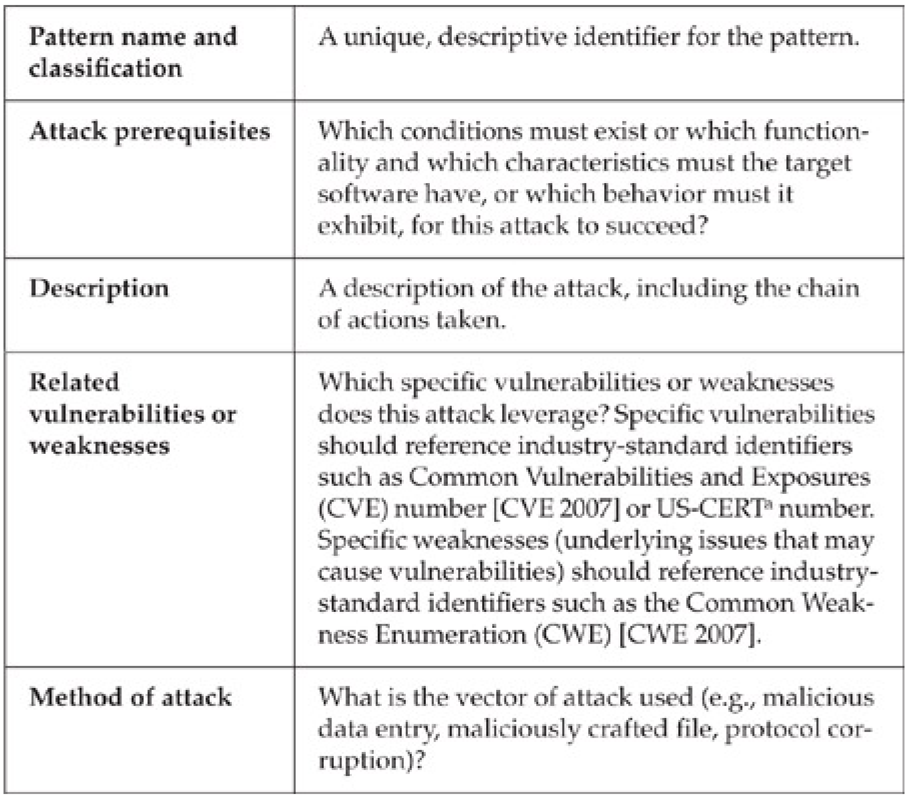
http://measurablesecurity.mitre.org/

What Does an Attack Pattern Look Like?

An attack pattern at a minimum should fully describe what the attack looks like, what sort of skill or resources are required to successfully execute it, and in which contexts it is applicable and should provide enough information to enable defenders to effectively prevent or mitigate it.

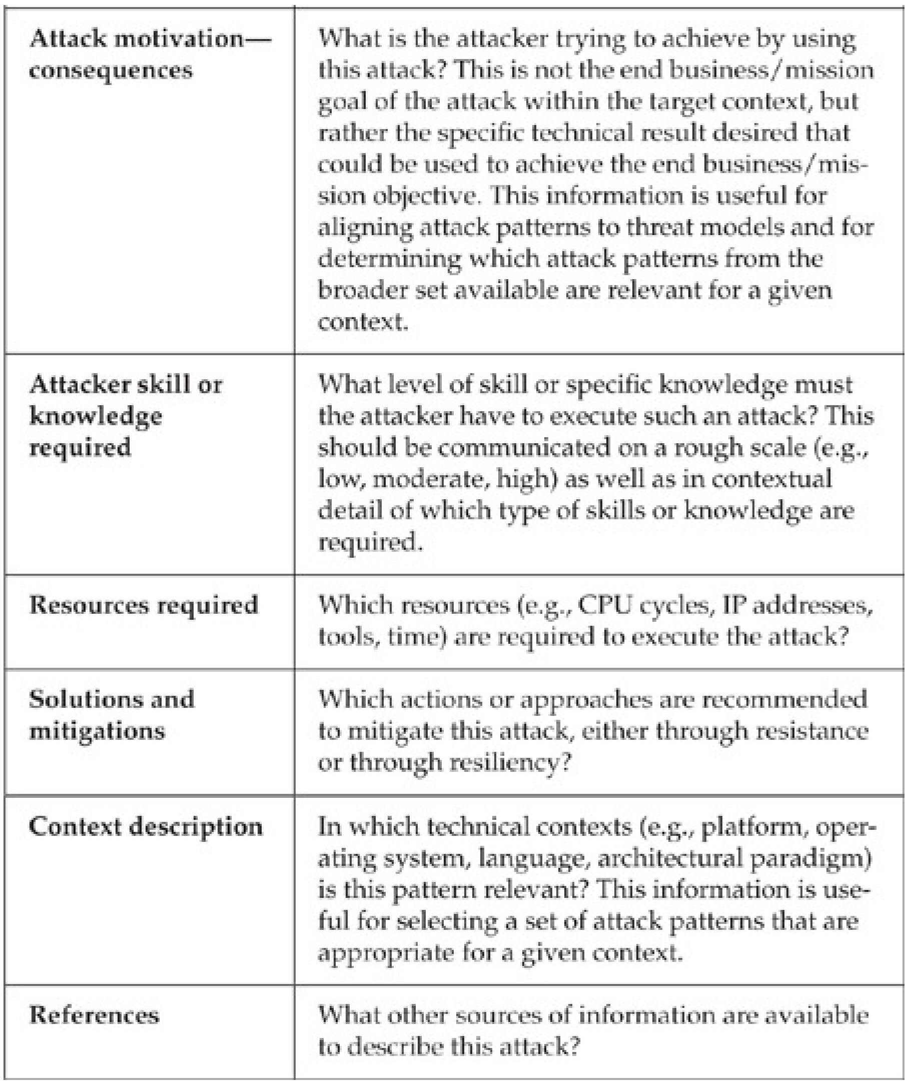
We propose that a simple attack pattern should typically include the information shown in **Table 2–1**.

Table 2-1. *Attack Pattern Components*



Pattern name and classification	A unique, descriptive identifier for the pattern.
Attack prerequisites	Which conditions must exist or which functionality and which characteristics must the target software have, or which behavior must it exhibit, for this attack to succeed?
Description	A description of the attack, including the chain of actions taken.
Related vulnerabilities or weaknesses	Which specific vulnerabilities or weaknesses does this attack leverage? Specific vulnerabilities should reference industry-standard identifiers such as Common Vulnerabilities and Exposures (CVE) number [CVE 2007] or US-CERT[a] number. Specific weaknesses (underlying issues that may cause vulnerabilities) should reference industry-standard identifiers such as the Common Weakness Enumeration (CWE) [CWE 2007].
Method of attack	What is the vector of attack used (e.g., malicious data entry, maliciously crafted file, protocol corruption)?

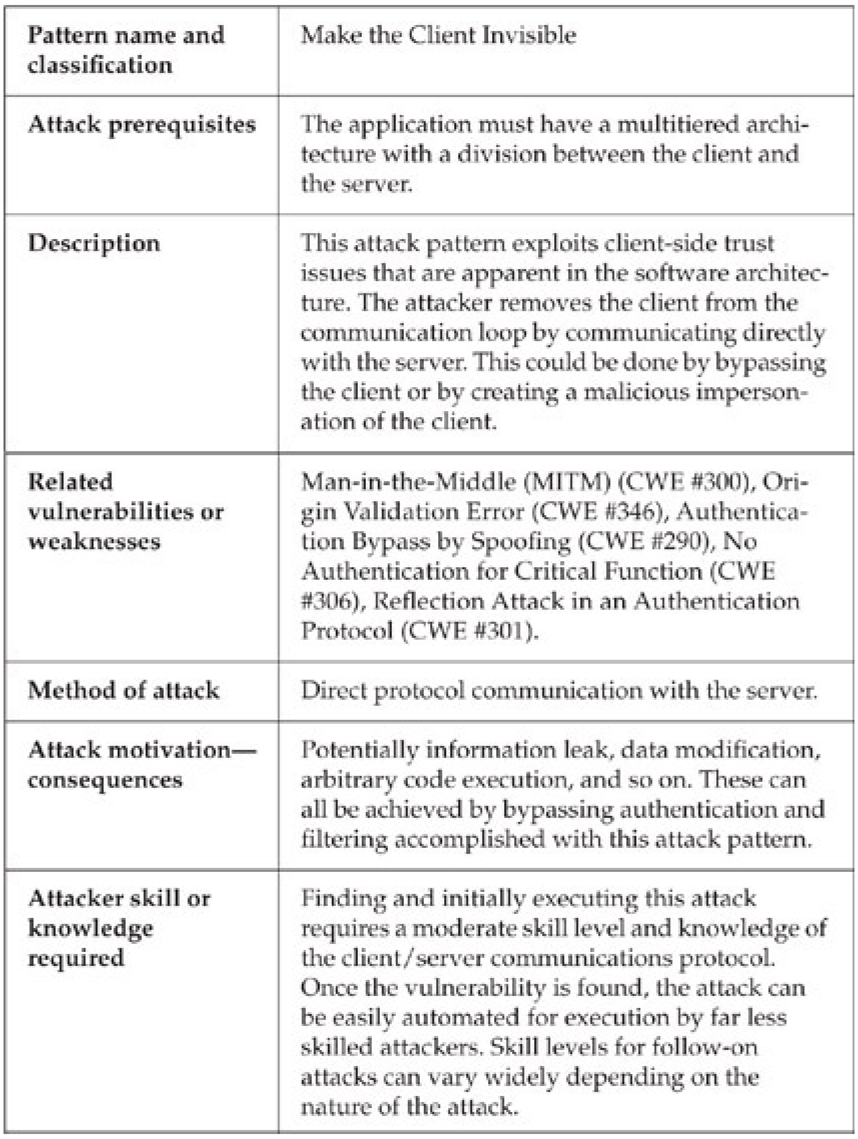
[a] http://www.us-cert.gov

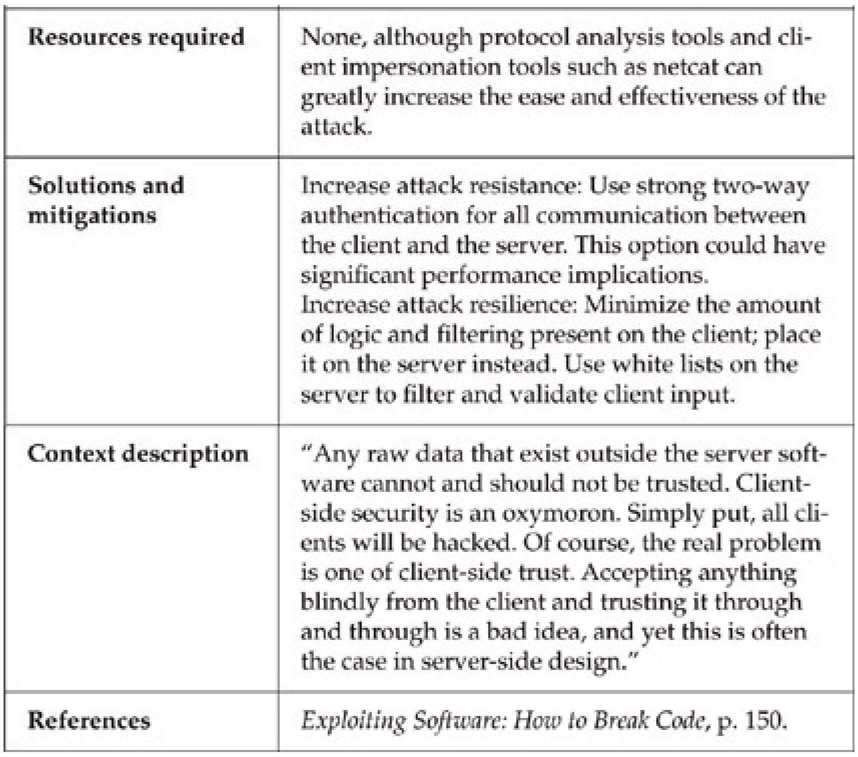


Attack motivation—consequences	What is the attacker trying to achieve by using this attack? This is not the end business/mission goal of the attack within the target context, but rather the specific technical result desired that could be used to achieve the end business/mission objective. This information is useful for aligning attack patterns to threat models and for determining which attack patterns from the broader set available are relevant for a given context.
Attacker skill or knowledge required	What level of skill or specific knowledge must the attacker have to execute such an attack? This should be communicated on a rough scale (e.g., low, moderate, high) as well as in contextual detail of which type of skills or knowledge are required.
Resources required	Which resources (e.g., CPU cycles, IP addresses, tools, time) are required to execute the attack?
Solutions and mitigations	Which actions or approaches are recommended to mitigate this attack, either through resistance or through resiliency?
Context description	In which technical contexts (e.g., platform, operating system, language, architectural paradigm) is this pattern relevant? This information is useful for selecting a set of attack patterns that are appropriate for a given context.
References	What other sources of information are available to describe this attack?

A simplified example of an attack pattern written to this basic schema is provided in Table 2–2. The idea for this pattern came from Hoglund and McGraw's book *Exploiting Software*, and a more detailed version is now available as CAPEC attack pattern #22.[9]

Table 2-2. *Example Attack Pattern*





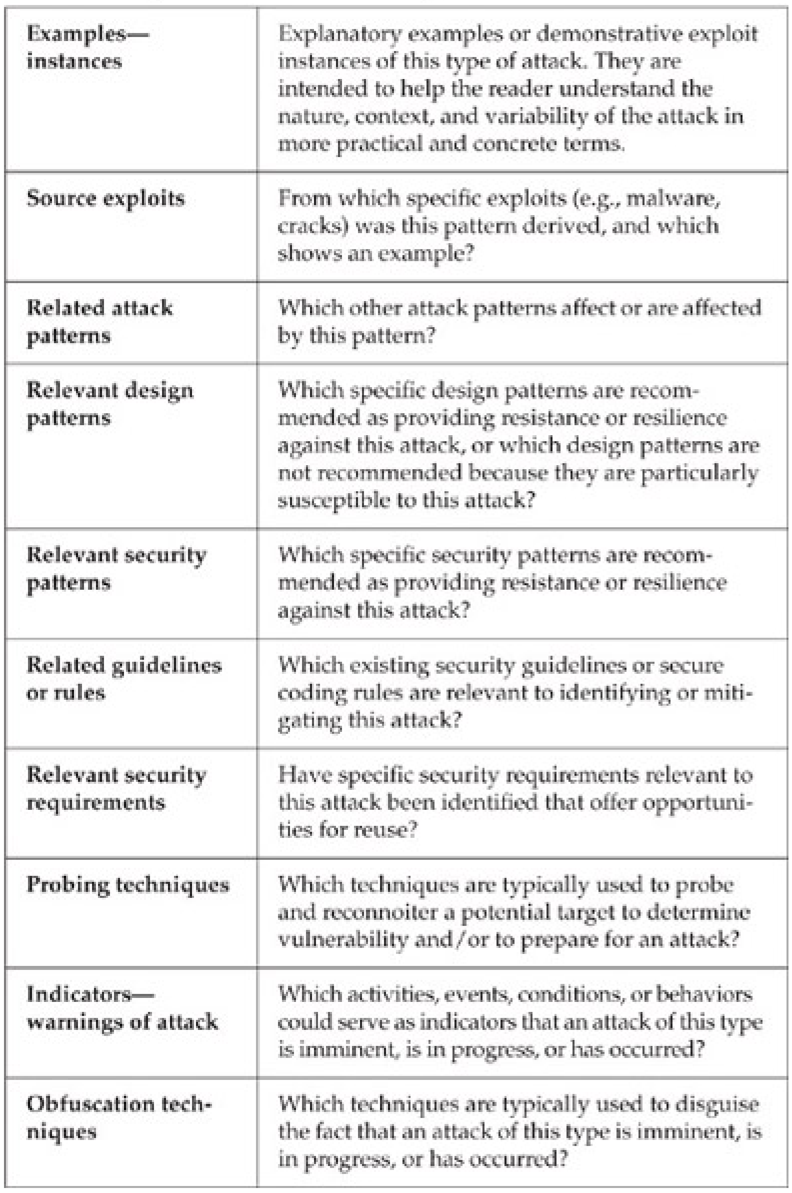
Pattern name and classification	Make the Client Invisible
Attack prerequisites	The application must have a multitiered architecture with a division between the client and the server.
Description	This attack pattern exploits client-side trust issues that are apparent in the software architecture. The attacker removes the client from the communication loop by communicating directly with the server. This could be done by bypassing the client or by creating a malicious impersonation of the client.
Related vulnerabilities or weaknesses	Man-in-the-Middle (MITM) (CWE #300), Origin Validation Error (CWE #346), Authentication Bypass by Spoofing (CWE #290), No Authentication for Critical Function (CWE #306), Reflection Attack in an Authentication Protocol (CWE #301).
Method of attack	Direct protocol communication with the server.
Attack motivation—consequences	Potentially information leak, data modification, arbitrary code execution, and so on. These can all be achieved by bypassing authentication and filtering accomplished with this attack pattern.
Attacker skill or knowledge required	Finding and initially executing this attack requires a moderate skill level and knowledge of the client/server communications protocol. Once the vulnerability is found, the attack can be easily automated for execution by far less skilled attackers. Skill levels for follow-on attacks can vary widely depending on the nature of the attack.
Resources required	None, although protocol analysis tools and client impersonation tools such as netcat can greatly increase the ease and effectiveness of the attack.
Solutions and mitigations	Increase attack resistance: Use strong two-way authentication for all communication between the client and the server. This option could have significant performance implications. Increase attack resilience: Minimize the amount of logic and filtering present on the client; place it on the server instead. Use white lists on the server to filter and validate client input.
Context description	"Any raw data that exist outside the server software cannot and should not be trusted. Client-side security is an oxymoron. Simply put, all clients will be hacked. Of course, the real problem is one of client-side trust. Accepting anything blindly from the client and trusting it through and through is a bad idea, and yet this is often the case in server-side design."
References	*Exploiting Software: How to Break Code*, p. 150.

Note that an attack pattern is not overly generic or theoretical. The following is not an attack pattern: “Writing outside array boundaries in an application can allow an attacker to execute arbitrary code on the computer running the target software.” This statement does not identify which type of functionality and specific weakness is targeted or how malicious input is provided to the application. Without that information, the statement is not particularly useful and cannot be considered an attack pattern.

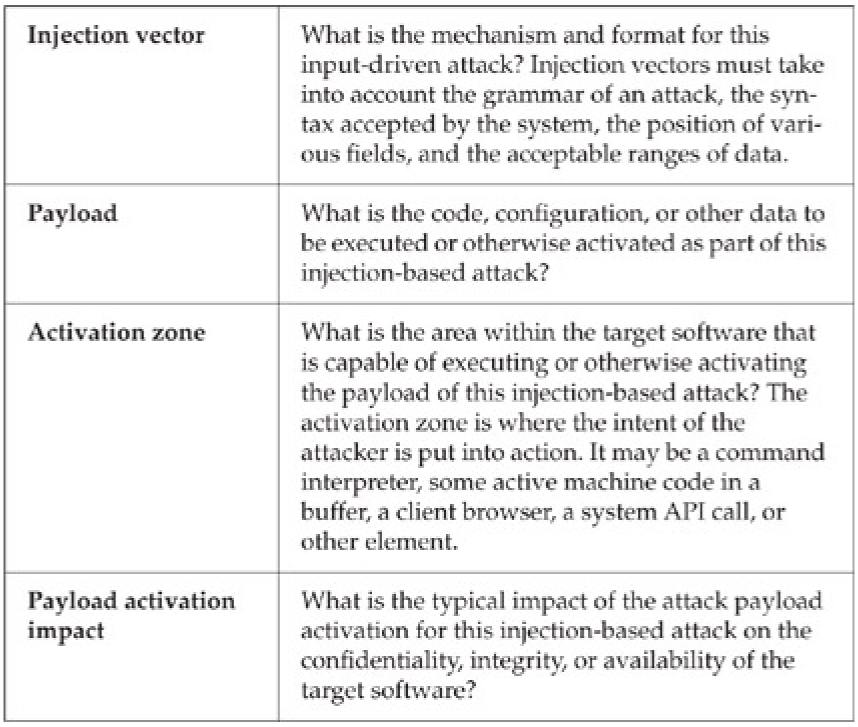
An attack pattern is also not an overly specific attack that applies only to a particular application, such as “When the PATH environment variable is set to a string of length greater than 128, the application foo executes the code at the memory location pointed to by characters 132, 133, 134, and 135 in the environment variable.” This amount of specificity is of limited benefit to the software development community because it does not help its members discover and fix vulnerabilities in other applications or even fix other similar vulnerabilities in the same application.

Although not broadly required or typical, it can be valuable to adorn attack patterns where possible and appropriate with other useful reference information such as that listed in **Table 2–3**.

Table 2-3. *Optional Attack Pattern Components*



Examples—instances	Explanatory examples or demonstrative exploit instances of this type of attack. They are intended to help the reader understand the nature, context, and variability of the attack in more practical and concrete terms.
Source exploits	From which specific exploits (e.g., malware, cracks) was this pattern derived, and which shows an example?
Related attack patterns	Which other attack patterns affect or are affected by this pattern?
Relevant design patterns	Which specific design patterns are recommended as providing resistance or resilience against this attack, or which design patterns are not recommended because they are particularly susceptible to this attack?
Relevant security patterns	Which specific security patterns are recommended as providing resistance or resilience against this attack?
Related guidelines or rules	Which existing security guidelines or secure coding rules are relevant to identifying or mitigating this attack?
Relevant security requirements	Have specific security requirements relevant to this attack been identified that offer opportunities for reuse?
Probing techniques	Which techniques are typically used to probe and reconnoiter a potential target to determine vulnerability and/or to prepare for an attack?
Indicators—warnings of attack	Which activities, events, conditions, or behaviors could serve as indicators that an attack of this type is imminent, is in progress, or has occurred?
Obfuscation techniques	Which techniques are typically used to disguise the fact that an attack of this type is imminent, is in progress, or has occurred?



Injection vector	What is the mechanism and format for this input-driven attack? Injection vectors must take into account the grammar of an attack, the syntax accepted by the system, the position of various fields, and the acceptable ranges of data.
Payload	What is the code, configuration, or other data to be executed or otherwise activated as part of this injection-based attack?
Activation zone	What is the area within the target software that is capable of executing or otherwise activating the payload of this injection-based attack? The activation zone is where the intent of the attacker is put into action. It may be a command interpreter, some active machine code in a buffer, a client browser, a system API call, or other element.
Payload activation impact	What is the typical impact of the attack payload activation for this injection-based attack on the confidentiality, integrity, or availability of the target software?

Leveraging Attack Patterns in All Phases of the Software Development Life Cycle

Unlike many of the defensive touchpoint activities and knowledge with a narrowly focused area of impact within the SDLC, attack patterns as a resource provide potential value to the development team during all phases of software development regardless of the SDLC chosen, including requirements, architecture, design, coding, testing, and even deploying the system.

Leveraging Attack Patterns in Positive and Negative Security Requirements

Security-focused requirements are typically split between positive requirements, which specify functional behaviors the software must exhibit (often security features), and negative requirements (typically in the form of misuse/abuse cases), which describe behaviors that the software must not exhibit if it is to operate securely.

Attack patterns can be an invaluable resource for helping to identify both positive and negative security requirements. They have obvious direct benefit in defining the software's expected reaction to the attacks they describe. When put into the context of the other functional requirements for the software and when considering the underlying weaknesses targeted by the attack, they can help identify both negative requirements describing potential undesired behaviors and positive functional requirements for avoiding—or at least mitigating—the potential attack. For instance, if a customer provides the requirement "The application must accept ASCII characters," then the attack pattern "Using Unicode Encoding to Bypass Validation Logic" (CAPEC #71)[10] can be used to ask the question, "What should the application do if Unicode characters or another unacceptable, non-ASCII character set is encountered?" From this question, misuse/abuse cases can be defined, such as "Malicious user provides Unicode characters to the data entry field." By having a specific definition for this negative requirement, the designers, implementers, and testers will have a clear idea of the type of hostile environment with which the software must deal and will build the software accordingly. This information can also help define positive requirements, such as "The system shall translate all input into the ASCII character set before processing that input." If these sorts of requirements are overlooked, the developed application may unknowingly accept Unicode characters in some instances, and an attacker could use that fact to bypass input filters for ASCII characters.

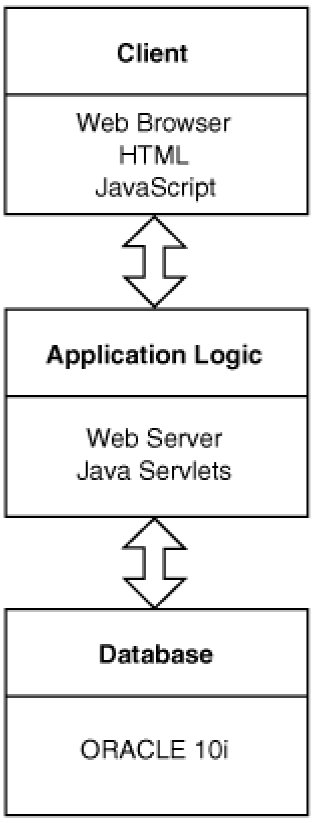
Many vulnerabilities result from vague specifications and requirements. In general, attack patterns allow the requirements engineer to ask "what if" questions in a structured and bounded way to make the requirements more specific. If an attack pattern states "Condition X can be leveraged by an attacker to cause Y," then a valid question may be "What should the application do if it encounters condition X?" Of course, one of the great challenges with any "what if" session is knowing when to stop. There is no hard-and-fast answer to this question, as it is very dependent on context. Using attack patterns, however, can help minimize this risk by offering a method to ask the appropriate "what if" questions within a defined rather than boundless scope.

Software security requirements as an element of software security engineering are discussed further in **Chapter 3**.

Leveraging Attack Patterns in Architecture and Design

Once requirements have been defined, all software must go through some level of architecture and design. Regardless of the formality of the process followed, the results of this activity will form the foundation for the software and drive all remaining development activities. During architecture and design, decisions must be made about how the software will be structured, how the various components will be integrated and interact, which technologies will be used, and how the requirements defining how the software will function will be interpreted. Careful consideration is necessary during this activity, given that as much as 50 percent of software defects leading to security problems are design flaws **[McGraw 2006]**. In the example depicted in **Figure 2–6**, a potential architecture could consist of a three-tier system with the client (a Web browser leveraging JavaScript/HTML), a Web server (leveraging Java servlets), and a database server (leveraging Oracle 10i). Decisions made at this level can have significant implications for the overall security profile of the software.

Figure 2-6. *Example architecture*



Attack patterns can be valuable during planning of the software's architecture and design in two ways. First, some attack patterns describe attacks that directly exploit architecture and design flaws in software. For instance, the Make the Client Invisible attack pattern (briefly described earlier in this chapter) exploits client-side trust issues that are apparent in the software architecture. Second, attack patterns at all levels can provide a useful context for the threats that the software is likely to face and thereby determine which architectural and design features to avoid or to specifically incorporate. The Make the Client Invisible attack pattern, for example, tells us that absolutely nothing sent back by the client can be trusted, regardless of which network security mechanisms (e.g., SSL) are used. The client is untrusted, and an attacker can send back literally any information that he or she desires. All input validation, authorization checks, and other security assessments must be performed on the server side. In addition, any data sent to the client should be considered visible by the client regardless of its intended presentation (that is, data that the client should not see should never be sent to the client). Performing authorization checks on the client side to determine which data to display is unacceptable.

The Make the Client Invisible attack pattern instructs the architects and designers that they must ensure that absolutely no security-critical business logic is performed on the client side. In fact, depending on the system requirements and the threats and risks faced by the system, the architects and designers may even want to define an input validator through which all input to the server must pass before being sent to the other classes. Such decisions must be made at the architecture and design phase, and attack patterns provide some guidance regarding what issues should be considered.

It is essential to document any attack patterns used in the architecture and design phase so that the application can be tested using those attack patterns. Tests must be created to validate that mitigations for the attack patterns considered during this phase were implemented properly.

Software architecture and design as an element of software security engineering is discussed further in **Chapter 4**.

Leveraging Attack Patterns in Implementation and Coding

If the architecture and design have been performed properly, each developer implementing the design should be writing well-defined components with well-defined interfaces.

Attack patterns can be useful during implementation because they enumerate the specific weaknesses targeted by relevant attacks and allow developers to ensure that these weaknesses do not occur in their code. These weaknesses could take the form of implementation bugs or simply valid coding constructs that can have security implications if used improperly. Unfortunately, implementation bugs are not always easy to avoid or to catch and fix. Even after applying basic review techniques, they can still remain abundant and can make software vulnerable to extremely dangerous exploits. It is important to extend basic review techniques by including more focused security-relevant concerns. Failure to properly check an array bound, for example, might permit an attacker to execute arbitrary code on the target host, whereas failure to perform proper input validation might enable an attacker to destroy an entire database.

Underlying security issues in non-buggy valid code are typically more difficult to identify. They cannot be identified with simple black-box scanning or testing, but instead require specialized knowledge of what these weaknesses look like. Here, we focus on how attack patterns can be used to identify specific weaknesses for targeting and mitigation through informing the developer ahead of time about those issues to avoid and through providing a list of issues (security coding rules) to look for in code reviews; the latter step can often be automated through the use of security scanning tools. It is important to determine precisely which attack patterns are applicable for a particular project. In some instances, different attack patterns may be applicable for different components of a product.

Good architecture and design, as well as developer awareness, enhanced with attack patterns can help to minimize security weaknesses. Nevertheless, it is also essential to ensure that all source code, once written, is reviewed with processes that have been shown to be capable of detecting the targeted weaknesses. Given the sometimes daunting size and sheer monotony of this task, it is typically performed using an automated analysis tool (e.g., those from Fortify, Ounce Labs, Klocwork, or Coverity). Even though analysis tools cannot find all security weaknesses, they can help weed out many potential issues. Using attack patterns as guidance, specific subsets of the tools' search rules can be selected and custom rules can be created for organizations to help find specific security weaknesses or instances of failure to follow security standards. For example, to deal with the Simple Script Injection (CAPEC #63)[11] attack pattern, an organization may establish a security standard in which all untrusted input is passed through an input filter and all output of data obtained from an untrusted source is passed through an encoder. An organization can develop a variety of such filters and encoders, and static source code analysis tools can help find occurrences in code where developers may have neglected to adhere to standards and opted to use Java's input/output features directly.

Software implementation and coding as an element of software security engineering is discussed further in Chapter 5.

Leveraging Attack Patterns in Software Security Testing

The testing phase differs from the previous phases in the SDLC in that its goal is not necessarily constructive; the goal of risk-based security testing is typically to break software so that the discovered issues can be fixed before an attacker can find them [Whittaker 2003]. The purpose of using attack patterns in this phase is to have the individuals performing the various levels and types of testing act as attackers attempting to break the software. In unit testing, applicable attack patterns should be used to identify relevant targeted weaknesses and to generate test cases for each component, thereby ensuring that each component avoids or at least resists these weaknesses. For example, to test for shell command injection using command delimiters, malicious input strings containing delimiter-separated shell commands should be crafted and input to the applicable component(s) to confirm that the software demonstrates the proper behavior when provided with this type of malicious data. In integration testing, a primary security issue to consider is whether the individual components make differing assumptions that affect security, such that the integrated whole may contain conflicts or ambiguities. Attack patterns documented in the architecture and design phase should be used to create integration tests exploring such ambiguities and conflicts.

In system testing, the entire system is exercised and probed to ensure that it meets all of its functional and nonfunctional requirements. If attack patterns were used in the requirements-gathering phase to generate security requirements, system testing will have a solid foundation for identifying test cases that validate secure behavior. These security requirements should be tested during system testing. For example, the Using Unicode Encoding to Bypass Validation attack pattern can be used to generate test cases that ensure that the application behaves properly when provided with unexpected characters as input. Testers should input characters that the application is not supposed to accept to see how the application behaves under these conditions. The application's actual behavior when under attack should be compared with the desired behavior defined in the security requirements.

Even if security is considered throughout the SDLC when building software, and even if extensive testing is performed, vulnerabilities will likely still exist in the software after deployment, simply because no useful software is 100 percent secure **[Viega 2001]**. Software can be designed and developed to be extremely secure, but if it is deployed and operated in an insecure fashion, many vulnerabilities can be introduced. For example, a piece of software might provide strong encryption and proper authentication before allowing access to encrypted data, but if an attacker can obtain valid authentication credentials, he or she can subvert the software's security. Nothing is 100 percent secure, which means that the environment must always be secured and monitored to thwart attacks.

Given these caveats, it is extremely important to perform security testing of the software in its actual operational environment. Vulnerabilities present in software can sometimes be masked by environmental protections such as network firewalls and application firewalls, and environmental conditions can sometimes create new vulnerabilities. Such issues can often be discovered using a mix of white-box and black-box analysis of the deployed environment. White-box analysis of deployed software involves performing security analysis of the software, including its deployed environment, with knowledge of the architecture, design, and implementation of the software. Black-box analysis (typically in the form of penetration testing) involves treating the deployed software as a "black box" and attempting to attack it without any knowledge of its inner workings. Black-box testing is good for finding the specific implementation issues you know to look for, whereas detailed and structured white-box testing can uncover unexpected architecture and design and implementation issues that you may not have known to look for. Both types of testing are important, and attack patterns can be leveraged for both.

Black box testing of Web applications is generally performed using tools such as application security testers like those from companies such as Watchfire that automatically run predefined tests. Attack patterns can be used as models to create the tests (simulated attacks) these tools perform, thereby giving them more significant relevance and effectiveness. These tools typically test for a large variety of attacks, but they generally cannot find subtle architectural vulnerabilities. Although they may effectively identify vulnerabilities that script kiddies and other relatively unskilled attackers would likely exploit, a skilled attacker would be able to find many issues that a vulnerability scanning tool simply could not detect. For instance, a lack of encryption for transmitting Social Security numbers would not be detected using an automated tool because the fact that Social Security numbers are unencrypted is not a purely technical flaw. The black-box testing tool cannot determine which information is a Social Security number and cannot apply business logic. Attack patterns that are useful for creating black-box tests include those that can be executed remotely without requiring many steps.

White-box testing is typically more thorough than black-box testing. It involves extensive analysis performed by security experts who have access to the software's requirements, architecture, design, and code. Because of the deeper understanding of the code involved, white-box security testing is often capable of finding more obscure implementation bugs that are not uncovered in black-box testing and, occasionally, some architecture and design flaws. Attack patterns can be leveraged to determine areas posing system risks, which can then be scrutinized by the system white-box analysis. Attack patterns that are effective guides for white-box analysis include those that focus on architecture and design weaknesses or implementation weaknesses. For example, an attack pattern that could be used in white-box testing of a deployed system is sniffing sensitive data on an insecure channel. Those with knowledge of data sensitivity classifications and an understanding of the business context around various types of data can determine whether some information that should always be communicated over an encrypted channel is actually being sent over an insecure channel. Such issues are often specific to a deployed environment; thus, analysis of the actual deployed software is required.

Software testing as an element of software security engineering is discussed further in **Chapter 5**.

M L L2

2.4. How to Assert and Specify Desired Security Properties[12]

Identifying and describing the properties that determine the security profile of software gave us the common language and objectives for building secure software. Outlining mechanisms for how these properties can be influenced gave us the ability to take action and effect positive change in regard to the security assurance of the software we build. Taken in combination, these achievements lay a foundation for understanding what makes software secure. Unfortunately, without a mechanism for clearly communicating the desired or attained security assurance of software in terms of these properties and activities, this understanding is incomplete. What is needed is a mechanism for asserting and specifying desired security properties and using them as a basis for planning, communicating, and assuring compliance. These assertions and specifications are typically captured and managed in an artifact known as an **assurance case**.

Elsewhere in this book and on the BSI Web site, you can learn about best practices, tools, and techniques that can help in building security into software. Nevertheless, the mere existence or claimed use of one or more of these best practices, tools, or techniques does not constitute an adequate assurance case. For example, in support of an overarching security claim (e.g., that a system is acceptably secure), security assurance cases must provide evidence that particular best practices, tools, and techniques were properly applied and must indicate by whom they were applied and how extensive their coverage is. Moreover, unlike many product certifications that quickly grow stale because they are merely snapshots in time of an infrequently applied certification process, a security assurance case should provide evidence that the practices, tools, or techniques being used to improve security were actually applied to the currently released version of the software (or that the results were invariant to any of the code changes that subsequently occurred).

2.4.1. Building a Security Assurance Case

A security assurance case uses a structured set of arguments and a corresponding body of evidence to demonstrate that a system satisfies specific claims with respect to its security properties. This case should be amenable to review by a wide variety of stakeholders. Although tool support is available for development of these cases, the creation and documentation of a security assurance case can be a demanding and time-consuming process. Even so, similarities may exist among security cases in the structure and other characteristics of the claims, arguments, and evidence used to construct them. A catalog of patterns (templates) for security assurance cases can facilitate the process of creating and documenting an individual case. Moreover, assurance case patterns offer the benefits of reuse and repeatability of process, as well as providing some notion of coverage or completeness of the evidence.

A security assurance case[13] is similar to a legal case, in that it presents arguments showing how a top-level claim (e.g., “The system is acceptably secure”) is supported by objective evidence. Unlike a typical product certification, however, a security case considers people and processes as well as technology. A case is developed by showing how the top-level claim is supported by subclaims. For example, part of a security assurance case would typically address various sources of security vulnerabilities. The case would probably claim that a system has none of the common coding defects that lead to security vulnerabilities, including, for example, buffer overflow vulnerabilities.[14] A subclaim about the absence of buffer overflow vulnerabilities could be supported by showing that (1) developers received training on how to write code that minimizes the possibility of buffer overflow vulnerabilities; (2) experienced developers reviewed the code to see if any buffer overflow possibilities existed and found none; (3) a static analysis tool scanned the code and found no problems; and (4) the system and its components were tested with invalid arguments and all such inputs were rejected or properly handled as exceptions.

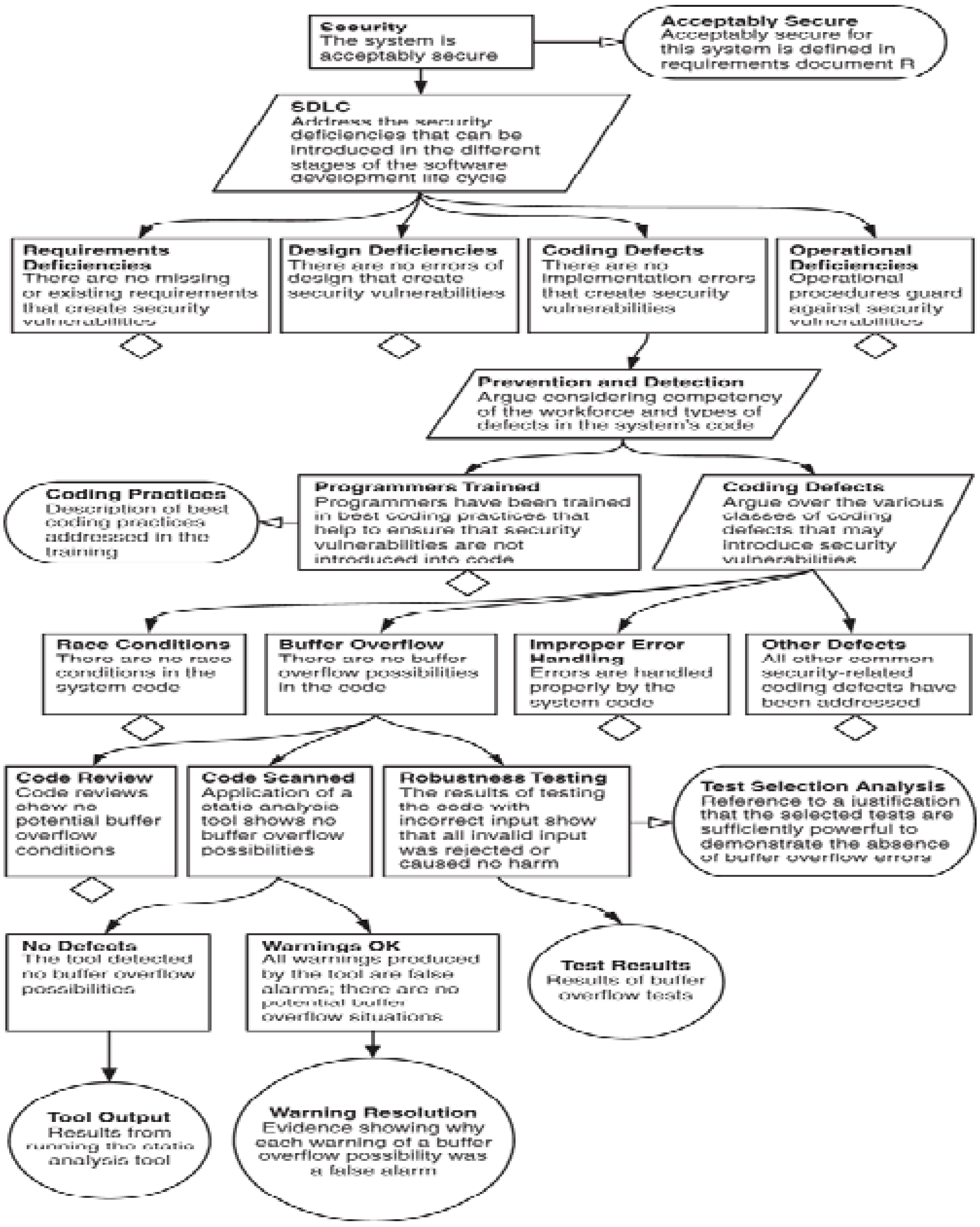
In this example, the “evidence” would consist of developer training credentials, the results of the code review, the output of the code scanner, and the results of the invalid-input tests. The “argument” is stated as follows: “Following best coding practice has value in *preventing* buffer overflow coding defects. Each of the other methods has value in *detecting* buffer overflow defects; none of them detected such defects (or these defects were corrected[15]), and so the existing evidence supports the claim that there are no buffer overflow vulnerabilities.”[16] Further information could show that this claim is incorrect.

Our confidence in the argument (that is, in the soundness of the claim) depends on how convincing we find the argument and the evidence. Moreover, if we believe that the consequences of an invalid claim are sufficiently serious, we might require that further evidence or other subclaims be developed. The seriousness of a claim would depend on the potential impact of an attack (e.g., projected economic loss, injury, or death) related to that claim and on the significance of the threat of such an attack. Although an in-depth discussion of the relation of threat and impact to security cases is beyond the scope of this book, a comprehensive security case should include—or should at least be developed in the context of—analyses of the threats to a system and the projected outcomes of successful attacks.

2.4.2. A Security Assurance Case Example

The structure for a partially developed security assurance case focusing on buffer overflow coding defects appears in **Figure 2–7**. This case is presented in a graphical notation called Goal Structuring Notation (GSN) **[Kelly 2004]**.

Figure 2-7. *Partially expanded security assurance case that focuses on buffer overflow*


Security The system is acceptably secure
Acceptably Secure Acceptably secure for this system is defined in requirements document R
SDLC Address the security deficiencies that can be introduced in the different stages of the software development life cycle
Requirements Deficiencies There are no missing or existing requirements that create security vulnerabilities
Design Deficiencies There are no errors of design that create security vulnerabilities
Coding Defects There are no implementation errors that create security vulnerabilities
Operational Deficiencies Operational procedures guard against security vulnerabilities
Prevention and Detection Argue considering competency of the workforce and types of defects in the system's code
Coding Practices Description of best coding practices addressed in the training
Programmers Trained Programmers have been trained in best coding practices that help to ensure that security vulnerabilities are not introduced into code
Coding Defects Argue over the various classes of coding defects that may introduce security vulnerabilities
Race Conditions There are no race conditions in the system code
Buffer Overflow There are no buffer overflow possibilities in the code
Improper Error Handling Errors are handled properly by the system code
Other Defects All other common security-related coding defects have been addressed
Code Review Code reviews show no potential buffer overflow conditions
Code Scanned Application of a static analysis tool shows no buffer overflow possibilities
Robustness Testing The results of testing the code with incorrect input show that all invalid input was rejected or caused no harm
Test Selection Analysis Reference to a justification that the selected tests are sufficiently powerful to demonstrate the absence of buffer overflow errors
No Defects The tool detected no buffer overflow possibilities
Warnings OK All warnings produced by the tool are false alarms; there are no potential buffer overflow situations
Test Results Results of buffer overflow tests
Tool Output Results from running the static analysis tool
Warning Resolution Evidence showing why each warning of a buffer overflow possibility was a false alarm

The case starts with a claim (in the shape of a rectangle) that “The system is acceptably secure.” To the right, a box with two rounded sides, labeled “Acceptably Secure,” provides *context* for the claim. This element of the case provides additional information on what it means for the system to be “acceptably” secure. For example, the referenced document might cite Health Information Portability and Accountability Act (HIPAA) requirements as they apply to a particular system, or it might classify the kinds of security breaches that would lead to different levels of loss (laying the basis for an expectation that more effort will be spent to prevent the more significant losses).

Under the top-level claim is a parallelogram labeled “SDLC.” This element shows the *strategy* to be used in developing an argument supporting the top-level claim and provides helpful insight to anyone reviewing the case.

In this example, the strategy is to address potential security vulnerabilities arising at the different stages of the SDLC—namely, requirements, design, implementation (coding), and operation.[17] One source of deficiencies is coding defects, which is the topic of one of the four subclaims. The other subclaims cover requirements, design, and operational deficiencies. (The diamond under a claim indicates that further expansion is required to fully elaborate the claim–argument–evidence substructure.) The structure of the argument implies that if these four subclaims are satisfied, the system is acceptably secure.

The strategy for arguing that there are no coding defects involves addressing actions taken both to *prevent* and to *detect* possible vulnerabilities caused by coding defects.[18] In **Figure 2–7**, only one possible coding defect—buffer overflow—is developed. Three types of evidence are developed to increase our confidence that no buffer overflow vulnerabilities are present, where each type of evidence is associated with each of three subclaims. The "Code Scanned" subclaim asserts that static analysis of the code has demonstrated the absence of buffer overflow defects. Below it are the subclaims that the tool definitively reported "No Defects" and that all warnings reported by the tool were subsequently verified as false alarms (that is, "Warnings OK"). Below these subclaims are two pieces of evidence: the tool output, which is the result of running the static analysis tool, and the resolution of each warning message, showing why each was a false alarm.

This is not a complete exposition of GSN. For instance, two other symbols, not shown in **Figure 2–7**, represent *justification* and *assumption*. As with the context element, they are used to provide additional information helpful in understanding the claim.

The claims in the example are primarily product focused and technical; that is, the claims address software engineering issues. An assurance case may also require taking into account legal, regulatory, economic (e.g., insurance), and other nontechnical issues **[Lipson 2002]**. For example, a more complete case might contain claims reflecting the importance of legal or regulatory requirements relating to the Sarbanes–Oxley Act or HIPAA. In addition, an analysis of the threat and consequences of security breaches will determine how much effort is put into developing certain claims or types of argument. If a security breach can lead to a major regulatory fine, the case may require a higher standard of evidence and argumentation than if a breach carries little economic penalty.

2.4.3. Incorporating Assurance Cases into the SDLC

Developing a security assurance case is not a trivial matter. In any real system, the number of claims involved and the amount of evidence required will be significant. The effort involved is offset by an expected decrease in effort required to find and fix security-related problems at the back end of product development and by a reduced level of security breaches and their attendant costs.

Creating and evolving the security case as the system is being developed is highly recommended. Developing even the preliminary outlines of an assurance case as early as possible in the SDLC can improve the development process by focusing attention on what needs to be assured and which evidence needs to be developed at each subsequent stage of the SDLC. Attempting to gather or generate the necessary security case evidence once development is complete may be not just much more costly, but simply impossible.

For maximum utility, a security assurance case should be a document that changes as the system it documents changes. That is, the case should take on a different character as a project moves through its life cycle. In the predevelopment stage, the case focuses on demonstrating the following points:

• The plan for a security case is appropriate for the security requirements of the proposed system.

• The technical proposals are appropriate for achieving the security requirements of the proposed system.

• It will be possible to demonstrate that security has been achieved during the project.

At development time, the following steps are taken in relation to the security assurance case (which is derived from the predevelopment case):

• It is updated with the results of all activities that contribute to the security evaluation (including evidence and argumentation) so that, by the time of deployment, the case will be complete.

• It is presented at design (and other) reviews and the outcomes are included in the case.

Using a configuration control mechanism to manage the security case will ensure its integrity, as well as help the case remain relevant to the project's development status.

Security cases provide a structured framework for evaluating the impact of changes to the system and can help ensure that the changes do not adversely impact security. The case should continue to be maintained after deployment of the system, especially whenever the system is modified. Examining the current case can help determine whether modifications will invalidate or change arguments and claims and, if so, will help identify the appropriate parts of the case that need to be updated. In addition, if parts of the system prove insecure even in the face of a well-developed case, it is important to understand why this particular chain of evidence–argument–claim reasoning was insufficient.

2.4.4. Related Security Assurance and Compliance Efforts

Security-Privacy Laws and Regulations

Laws and regulations such as Sarbanes–Oxley and HIPAA mandate specific security and privacy requirements. Security assurance cases can be used to argue that a corporation is in compliance with a given law or regulation. One can envision the development of security case patterns for particular laws or regulations to assist in demonstrating such compliance.

Common Criteria

The Common Criteria (CC) is an internationally recognized standard for evaluating security products and systems **[CCMB 2005a, 2005b]**.

Protection profiles represent sets of security requirements that products can be evaluated and certified against. The results of a CC evaluation include an Evaluation Assurance Level (EAL), which indicates the strength of assurance. Although a CC evaluation includes elements that are similar to those found in a security case, the security case is a more general framework into which the results of CC evaluations can be placed as evidence of assurance. Anyone creating a product or system meant to satisfy a protection profile needs a way to argue that it does, in fact, match the requirements of the profile. Unlike ad hoc approaches to arguing about the achievement of certain security levels, the security case method provides an organizing structure and a common "language" that can be used to make assurance arguments about satisfying the set of requirements in a protection profile (at a particular EAL), as well as providing a broader framework that can be used to place CC evaluations in the context of other available evidence of assurance.

The standard format of CC evaluations allows for reuse of some of the basic elements in an assurance argument and hence may be thought of as providing patterns of evaluation. For example, the Common Criteria provides catalogs of standard Security Functional Requirements and Security Assurance Requirements. In contrast, security case patterns allow for the reuse of entire claim–argument–evidence structures and are, therefore, patterns in a much more general sense. Unlike CC evaluations, a security case is well suited to be maintained over time as a system development artifact. Thus the assurance case could evolve along with the system, always reflecting the system's current state and configuration.

2.4.5. Maintaining and Benefitting from Assurance Cases

Assurance cases for security provide a structured and reviewable set of artifacts that make it possible to demonstrate to interested parties that the system's security requirements have been met to a reasonable degree of certainty.[19] Moreover, the creation of an assurance case can help in the planning and conduct of development. The process of maintaining an assurance case can help developers identify new security issues that may arise when changes are made to the system. Developing and maintaining security cases throughout the SDLC is an emerging area of best practice for systems with critical security requirements.

A key difference between arguments related to security and arguments related to other quality attributes of a system is the presence of an intelligent adversary. Intelligent adversaries do not follow predictable courses, but rather try to attack where you least expect. Having an intelligent adversary implies that security threats will evolve and adapt. As a consequence, a security case developed today may have its assumptions unexpectedly violated, or its strength may not be adequate to protect against the attack of tomorrow. This evolutionary nature of threats suggests that security assurance cases will need to be revisited more frequently than assurance cases for safety, reliability, or other dependability properties.

One should not think of the creation, use, sharing, and evolution of security cases as a method that is in competition with other security certification or evaluation methods, tools, or techniques. Security cases provide a general framework in which to incorporate and integrate existing and future certification and evaluation methods into a unified argument and evidentiary structure. The security case is particularly valuable as a supporting framework because it allows you to make meta-arguments about the methods, tools, and techniques being used to establish assurance. For example, a security case might argue that a certification method applied by a third-party certifier provides higher assurance than the same method applied by the vendor of the product being certified. This type of meta-argument is outside the scope of the certification method itself.

Although further research and tool development are certainly needed, you can take advantage of the assurance case method right now. There is much to be gained by integrating even rudimentary security cases and security case patterns into the development life cycle for any mission-critical system. Even a basic security case is a far cry above the typical ad hoc arguments and unfounded reassurances in its ability to provide a compelling argument that a desired security property has been built into a system from the outset and has continued to be maintained throughout the SDLC.

E M L

2.5. Summary

As a project manager looking for understanding and guidance on building better security into your software, it is first crucial that you understand which characteristics of software make it more or less secure.

Three areas of knowledge and practice were recommended in this chapter:

• A solid understanding of the core security properties (confidentiality, integrity, availability, accountability, and non-repudiation) and of the other properties that influence them (dependability, correctness, predictability, reliability, safety, size, complexity, and traceability) provides a solid foundation for communicating software security issues and for understanding and placing into context the various activities, resources, and suggestions discussed in this book.

• Understanding both the defensive and attacker's perspectives as well as activities (touchpoints) and resources (attack patterns and others) available to influence the security properties of software can enable you to place the various activities, resources, and suggestions described in this book into effective action and cause positive change.

• Assurance cases provide a powerful tool for planning, tracking, asserting, assessing, and otherwise communicating the claims, arguments, and evidence (in terms of security properties, perspectives, activities, and resources) for the security assurance of software.

Understanding the core properties that make software secure, the activities and knowledge available to influence them, and the mechanisms available to assert and specify them lays the foundation for the deeper discussion of software security practices and knowledge found in the following chapters. As you explore each life-cycle phase or concern and examine each discussed practice or knowledge resource, it may be beneficial for you to revisit this chapter, using the content here as a lens to understand and assess the practice or resource for value and applicability in your own unique context.