UNIT-1

NEED OF SOFTWARE SECURITY AND LOW-LEVEL ATTACKS

Software Assurance and Software Security – threats to software security – Sources of Insecurity – Benefits of Detecting Software Security – Properties of Secure Software - Memory based Attacks: Low-Level Attacks against Heap and Stack – Defence against Memory based Attacks

Introduction

Software is everywhere. We all rely on complex, interconnected, software intensive information systems that use the internet as their means for communicating and transporting information. Building, deploying, operating and using software that has not been developed with high security in mind can be high risk. Here we discusses why security is increasingly a software problem. It defines the dimensions of software assurance and software security. It identifies threats that target most software and the shortcomings of the software development process that can render software vulnerable to threats.

SOFTWARE ASSURANCE AND SOFTWARE SECURITY

Software assurance has become critical because dramatic increases in business and mission risks are now known to be attributable to exploitable software.

The growing extent of the resulting risk exposure is rarely understood, as evidenced by these facts:

- Software is the weakest link in the successful execution of interdependent systems and software applications.
- Software size and complexity obscure intent and preclude exhaustive testing.
- Outsourcing and the use of unvetted software supply-chain components increase risk exposure.
- The sophistication and increasingly more stealthy nature of attacks facilitates exploitation.
- Reuse of legacy software with other applications introduces unintended consequences, increasing the number of vulnerable targets.
- Business leaders are unwilling to make risk-appropriate investments in software security.

According to the U.S. Committee on National Security Systems' **"National**

Information Assurance (IA) Glossary”, software assurance is the level of confidence that software is free from vulnerabilities, either intentionally designed into the software or accidentally inserted at any time during its life cycle, and that the software functions in the intended manner.

- Software assurance includes the disciplines of software reliability (also known as software fault tolerance), software safety, and software security.
- The focus of **Software Security Engineering: A Guide for Project Managers** is on the third of these, software security, which is the ability of software to resist, tolerate, and recover from events that intentionally threaten its dependability.
- The **main objective** of software security is to build more-robust, higher-quality, defect-free software that continues to function correctly under malicious attack.

The **objective of software security** is to field software-based systems that satisfy the following criteria:

- The system is as vulnerability and defect free as possible.
- The system limits the damage resulting from any failures caused by attack-triggered faults, ensuring that the effects of any attack are not propagated, and it recovers as quickly as possible from those failures.
- The system continues operating correctly in the presence of most attacks by either resisting the exploitation of weaknesses in the software by the attacker or tolerating the failures that result from such exploits.

Software that has been developed with security in mind generally reflects the following properties throughout its development life cycle:

Predictable execution. There is justifiable confidence that the software, when executed, functions as intended. The ability of malicious input to alter the execution or outcome in a way favorable to the attacker is significantly reduced or eliminated.

Trustworthiness. The number of exploitable vulnerabilities is intentionally minimized to the greatest extent possible. The goal is no exploitable vulnerabilities.

Conformance. Planned, systematic, and multidisciplinary activities ensure that software components, products, and systems conform to requirements and applicable standards and procedures for specified uses.

These objectives and properties must be interpreted and constrained based on the practical

realities that you face, such as what constitutes an adequate level of security, what is most critical to address, and which actions fit within the project's cost and schedule. ***These are risk management decisions.***

To achieve attack resilience, a software system should be able to recover from failures that result from successful attacks by resuming operation at or above some predefined minimum acceptable level of service in a timely manner. The system must eventually recover full service at the specified level of performance. These qualities and properties, as well as ***attack patterns.***

The Role of Processes and Practices in Software Security

- A number of factors influence how likely software is to be secure. For instance, **software vulnerabilities** can originate in the processes and practices used in its creation.
- These sources include the decisions made by software engineers, the flaws they introduce in specification and design, and the faults and other defects they include in developed code, inadvertently or intentionally.
- Other factors may include the choice of programming languages and development tools used to develop the software, and the configuration and behavior of software components in their development and operational environments.
- It is increasingly observed, however, that the most critical difference between secure software and insecure software lies in the nature of the processes and practices used to specify, design, and develop the software.
- The return on investment when security analysis and secure engineering practices are introduced early in the development cycle ranges from 12 percent to 21 percent, with the highest rate of return occurring when the analysis is performed during application design.
- This return on investment occurs because there are fewer security defects in the released product and hence reduced labor costs for fixing defects that are discovered later.
- A project that adopts a security-enhanced software development process is adopting a set of practices that initially should reduce the number of exploitable faults and weaknesses.
- Over time, as these practices become more codified, they should decrease the likelihood that such vulnerabilities are introduced into the software in the first place.

- More and more, research results and real-world experiences indicate that correcting potential vulnerabilities as early as possible in the software development life cycle, mainly through the adoption of security-enhanced processes and practices, is far more cost-effective than the currently pervasive approach of developing and releasing frequent patches to operational software.

THREATS TO SOFTWARE SECURITY

In information security, the threat—the source of danger—is often a person intending to do harm, using one or more malicious software agents. **Software is subject to two general categories of threats:**

Threats during development (mainly insider threats). A software engineer can sabotage the software at any point in its development life cycle through intentional exclusions from, inclusions in, or modifications of the requirements specification, the threat models, the design documents, the source code, the assembly and integration framework, the test cases and test results, or the installation and configuration instructions and tools.

Threats during operation (both insider and external threats). Any software system that runs on a network-connected platform is likely to have its vulnerabilities exposed to attackers during its operation. Attacks may take advantage of publicly known but unpatched vulnerabilities, leading to memory corruption, execution of arbitrary exploit scripts, remote code execution, and buffer overflows. Software flaws can be exploited to install spyware, adware, and other malware on users' systems that can lie dormant until it is triggered to execute.

Today, most project and IT managers responsible for system operation respond to the increasing number of Internet-based attacks by relying on operational controls at the operating system, network, and database or Web server levels while failing to directly address the insecurity of the application-level software that is being compromised.

This approach has two critical shortcomings:

1. The security of the application depends completely on the robustness of operational protections that surround it.
2. Many of the software-based protection mechanisms (controls) can easily be misconfigured or misapplied. Also, they are as likely to contain exploitable vulnerabilities as the application software they are (supposedly) protecting.

SOURCES OF SOFTWARE INSECURITY

- Most commercial and open-source applications, middleware systems, and operating systems are extremely large and complex. In normal execution, these systems can transition through a vast number of different states.
- These characteristics make it particularly difficult to develop and operate software that is consistently correct, let alone consistently secure.
- The unavoidable presence of security threats and risks means that project managers and software engineers need to pay attention to software security even if explicit requirements for it have not been captured in the software's specification.
- A large percentage of security weaknesses in software could be avoided if project managers and software engineers were routinely trained in how to address those weaknesses systematically and consistently.
- Unfortunately, these personnel are seldom taught how to design and develop secure applications and conduct quality assurance to test for insecure coding errors and the use of poor development techniques.
- They do not generally understand which practices are effective in recognizing and removing faults and defects or in handling vulnerabilities when software is exploited by attackers. They are often unfamiliar with the security implications of certain software requirements (or their absence). Likewise, they rarely learn about the security implications of how software is architected, designed, developed, deployed, and operated.
- The absence of this knowledge means that security requirements are likely to be inadequate and that the resulting software is likely to deviate from specified (and unspecified) security requirements.
- In addition, this lack of knowledge prevents the manager and engineer from recognizing and understanding how mistakes can manifest as exploitable weaknesses and vulnerabilities in the software when it becomes operational.

Software—especially networked, application-level software—is most often compromised by exploiting weaknesses that result from the following sources:

- Complexities, inadequacies, and/or changes in the software's processing model (e.g., a
- Web- or service-oriented architecture model).

- Incorrect assumptions by the engineer, including assumptions about the capabilities, outputs, and behavioral states of the software's execution environment or about expected inputs from external entities (users, software processes).
- Flawed specification or design, or defective implementation of
    - The software's interfaces with external entities. Development mistakes of this type include inadequate (or nonexistent) input validation, error handling, and exception handling.
    - The components of the software's execution environment (from middleware- level and operating-system-level to firmware- and hardware-level components).
- Unintended interactions between software components, including those provided by a third party.

Mistakes are unavoidable. Even if they are avoided during requirements engineering and design (e.g., through the use of formal methods) and development (e.g., through comprehensive code reviews and extensive testing), vulnerabilities may still be introduced into software during its assembly, integration, deployment, and operation.

No matter how faithfully a security-enhanced life cycle is followed, as long as software continues to grow in size and complexity, some number of exploitable faults and other weaknesses are sure to exist.

THE BENEFITS OF DETECTING SOFTWARE SECURITY DEFECTS EARLY

- Proactively tackling software security is often under-budgeted and dismissed as a luxury.
- In an attempt to shorten development schedules or decrease costs, software project managers often reduce the time spent on secure software practices during requirements analysis and design.
- In addition, they often try to compress the testing schedule or reduce the level of effort. Skimping on software quality is one of the worst decisions an organization that wants to maximize development speed can make; higher quality (in the form of lower defect rates) and reduced development time go hand in hand.
- The below figure illustrates the relationship between defect rate and

development time.

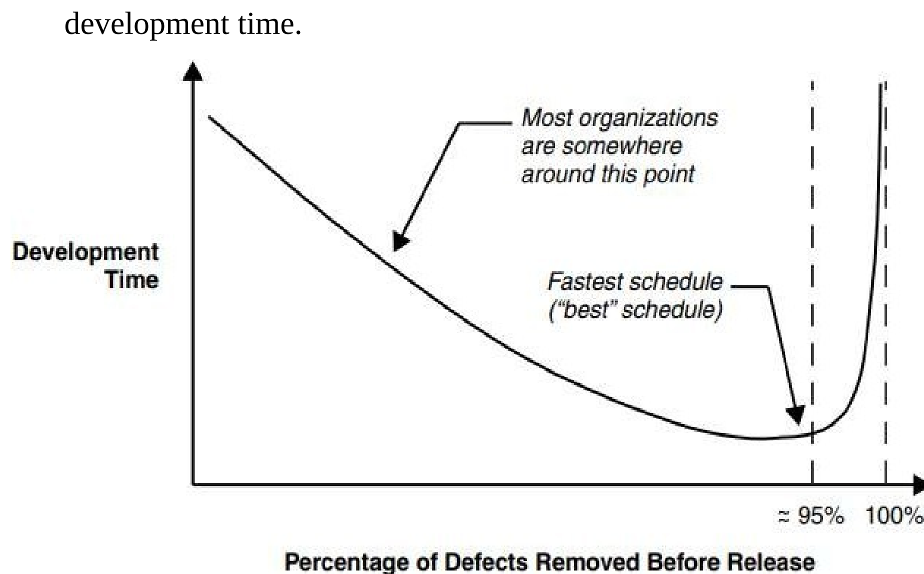


Figure : *Relationship between defect rate and development time*

- Projects that achieve lower defect rates typically have shorter schedules. But many organizations currently develop software with defect levels that result in longer schedules than necessary.
- The **"95 percent defect removal"** line is significant because that level of prerelease defect removal appears to be the point at which projects achieve the shortest schedules for the least effort and with the highest levels of user satisfaction.
- If more than 5 percent of defects are found after a product has been released, then the product is vulnerable to the problems associated with low quality, and the organization takes longer to develop its software than necessary.
- Projects that are completed with undue haste are particularly vulnerable to short changing quality assurance at the individual developer level. Any developer who has been pushed to satisfy a specific deadline or ship a product quickly knows how much pressure there can be to cut corners because "we're only three weeks from the deadline."
- As many as four times the average number of defects are reported for released software products that were developed under excessive schedule pressure. Developers participating in projects that are in schedule trouble often become obsessed with working harder rather than working smarter, which gets them into even deeper schedule trouble.
- One aspect of quality assurance that is particularly relevant during rapid development is the presence of error-prone modules—that is, modules that are

responsible for a disproportionate number of defects.

- They often are developed under excessive schedule pressure and are not fully tested. If development speed is important, then identification and redesign of error-prone modules should be a high priority.
- If an organization can prevent defects or detect and remove them early, it can realize significant cost and schedule benefits. Studies have found that reworking defective requirements, design, and code typically accounts for 40 to 50 percent of the total cost of software development.
- As a rule of thumb, every hour an organization spends on defect prevention reduces repair time for a system in production by three to ten hours. In the worst case, reworking a software requirements problem once the software is in operation typically costs 50 to 200 times what it would take to rework the same problem during the requirements phase. It is easy to understand why this phenomenon occurs.
- For example, a one-sentence requirement could expand into 5 pages of design diagrams, then into 500 lines of code, then into 15 pages of user documentation and a few dozen test cases. It is cheaper to correct an error in that one-sentence requirement at the time requirements are specified (assuming the error can be identified and corrected) than it is after design, code, user documentation, and test cases have been written.
- The below Figure illustrates that the longer defects persist, the more expensive they are to correct.

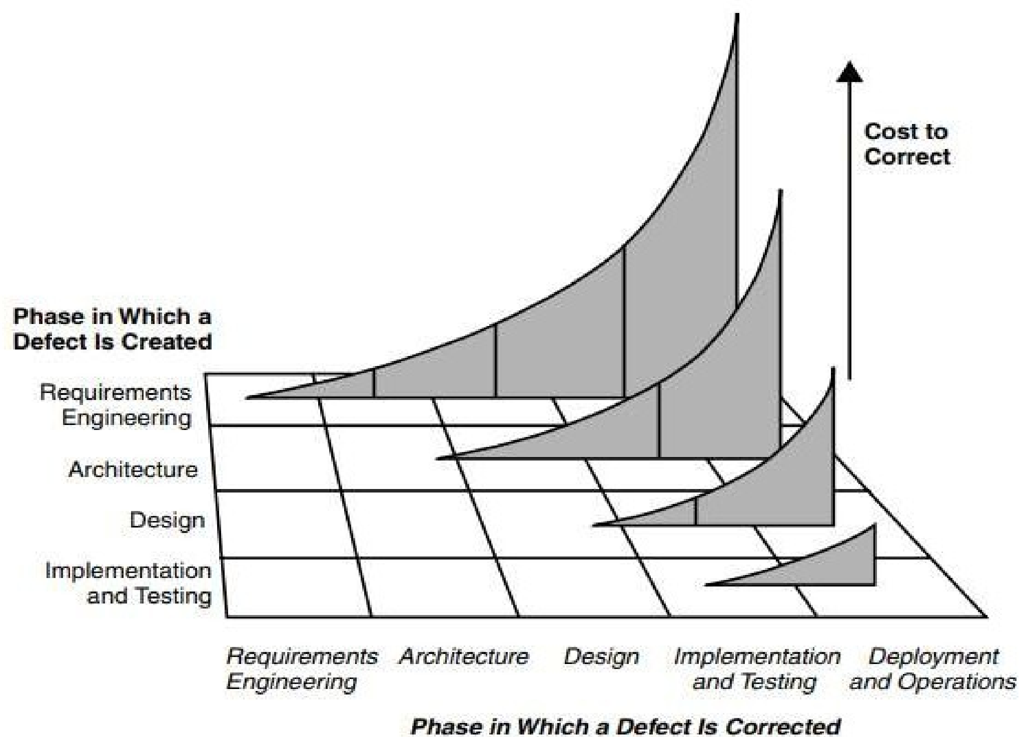


Figure : *Cost of correcting defects by life-cycle phase*

- The savings potential from early defect detection is significant: Approximately 60 percent of all defects usually exist by design time.
- A decision early in a project to exclude defect detection amounts to a decision to postpone defect detection and correction until later in the project, when defects become much more expensive and time-consuming to address.
- When a software product has too many defects, including security flaws, vulnerabilities, and bugs, software engineers can end up spending more time correcting these problems than they spent on developing the software in the first place.
- Project managers can achieve the shortest possible schedules with a higher-quality product by addressing security throughout the SDLC, especially during the early phases, to increase the likelihood that software is more secure the first time.

Making the business case for software security: current state

- As software project managers and developers, we know that when we want to introduce new approaches in our development processes, we have to make a cost–benefit argument to executive management to convince them that this move offers a business or strategic return on investment.

- Executives are not interested in investing in new technical approaches simply because they are innovative or exciting. For profit making organizations, we need to make a case that demonstrates we will improve market share, profit, or other business elements.
- For other types of organizations, we need to show that we will improve our software in a way that is important—in a way that adds to the organization's prestige, that ensures the safety of troops in the battle field, and so on.

Managing Secure Software Development

Which security strategy questions should I ask?

Achieving an adequate level of software security means more than complying with regulations or implementing commonly accepted best practices.

Consider the **following questions** from an enterprise perspective. Answers to these questions aid in understanding security risks to achieving project goals and objectives.

- What is the value we must protect?
- To sustain this value, which assets must be protected? Why must they be protected?
- What happens if they're not protected?
- What potential adverse conditions and consequences must be prevented and managed? At what cost? How much disruption can we stand before we take action?
- How do we determine and effectively manage residual risk (the risk remaining after mitigation actions are taken)?
- How do we integrate our answers to these questions into an effective, implementable, enforceable security strategy and plan?

The **answers to these questions** can help you determine how much to invest, where to invest, and how fast to invest in an effort to mitigate software security risk.

In the **absence of answers** to these questions (and a process for periodically reviewing and updating them), you (and your business leaders) will find it difficult to define and deploy an effective security strategy and, therefore, may be unable to effectively govern and manage enterprise, information, and software security.

A risk management framework for software security:

- A necessary part of any approach to ensuring adequate software security is the definition and use of a continuous risk management process. Software security

risk includes risks found in the outputs and results produced by each life-cycle phase during assurance activities, risks introduced by insufficient processes, and personnel-related risks.

- The risk management framework (RMF) introduced here and it can be used to implement a high-level, consistent, iterative risk analysis that is deeply integrated throughout the SDLC.
- The below Figure shows the RMF as a closed-loop process with five activity stages. Throughout the application of the RMF, measurement and reporting activities occur. These activities focus on tracking, displaying, and understanding progress regarding software risk.

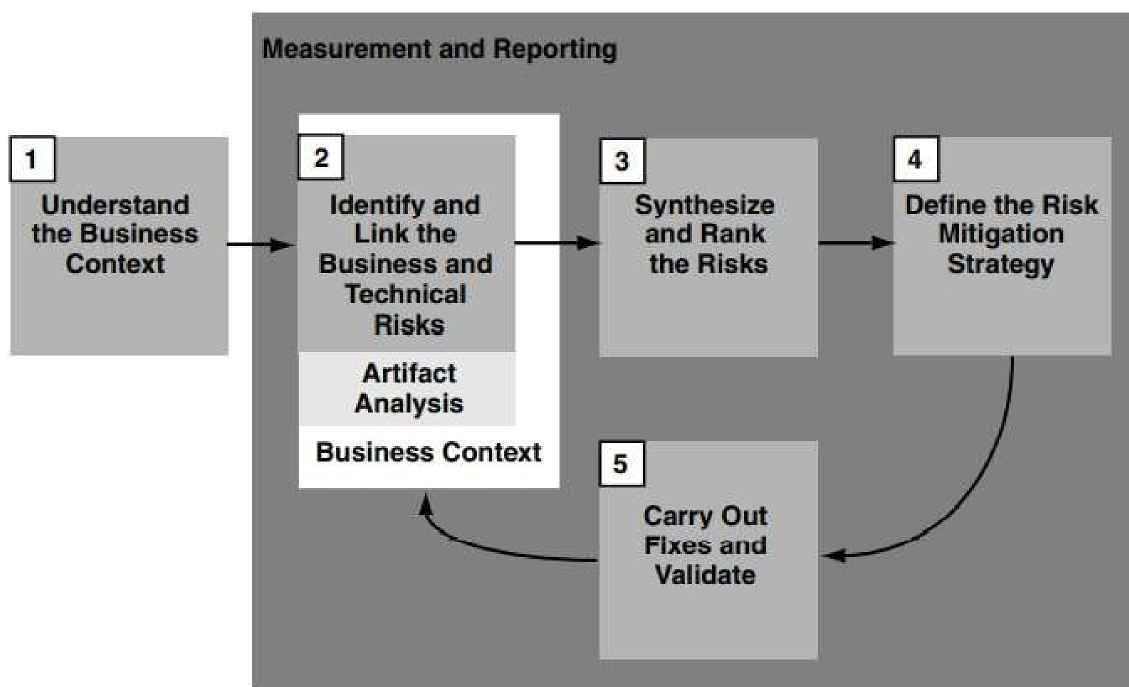


Figure : *A software security risk management framework*

<u>Software Security Practices in the Development Life Cycle:</u>

- Managers and software engineers should treat all software faults and weaknesses as potentially exploitable. Reducing exploitable weaknesses begins with the specification of software security requirements, along with considering requirements that may have been overlooked.
- Software that includes security requirements (such as security constraints on process behaviors and the handling of inputs, and resistance to and tolerance

of intentional failures) is more likely to be engineered to remain dependable and secure in the face of an attack.

- In addition, exercising misuse/abuse cases that anticipate abnormal and unexpected behavior can aid in gaining a better understanding of how to create secure and reliable software.
- Developing software from the beginning with security in mind is more effective by orders of magnitude than trying to validate, through testing and verification, that the software is secure. For example, attempting to demonstrate that an implemented system will never accept an unsafe input (that is, proving a negative) is impossible.
    - You can prove, however, using approaches such as formal methods and function abstraction, that the software you are designing will never accept an unsafe input.
- In addition, it is easier to design and implement the system so that input validation routines check every input that the software receives against a set of predefined constraints.
- Testing the input validation function to demonstrate that it is consistently invoked and correctly performed every time input enters the system is then included in the system's functional testing.
- Analysis and modeling can serve to better protect your software against the more subtle, complex attack patterns involving externally forced sequences of interactions among components or processes that were never intended to interact during normal software execution.
- Analysis and modeling can help you determine how to strengthen the security of the software's interfaces with external entities and increase its tolerance of all faults. Methods in support of analysis and modelling during each life-cycle phase such as attack patterns, misuse and abuse cases, and architectural risk analysis.
- If your development organization's time and resource constraints prevent secure development practices from being applied to the entire software system, you can use the results of a business-driven risk assessment to determine which software components should be given highest priority.
- A security-enhanced life-cycle process should (at least to some extent) compensate for security inadequacies in the software's requirements by adding

risk-driven practices and checks for the adequacy of those practices during all software life-cycle phases.

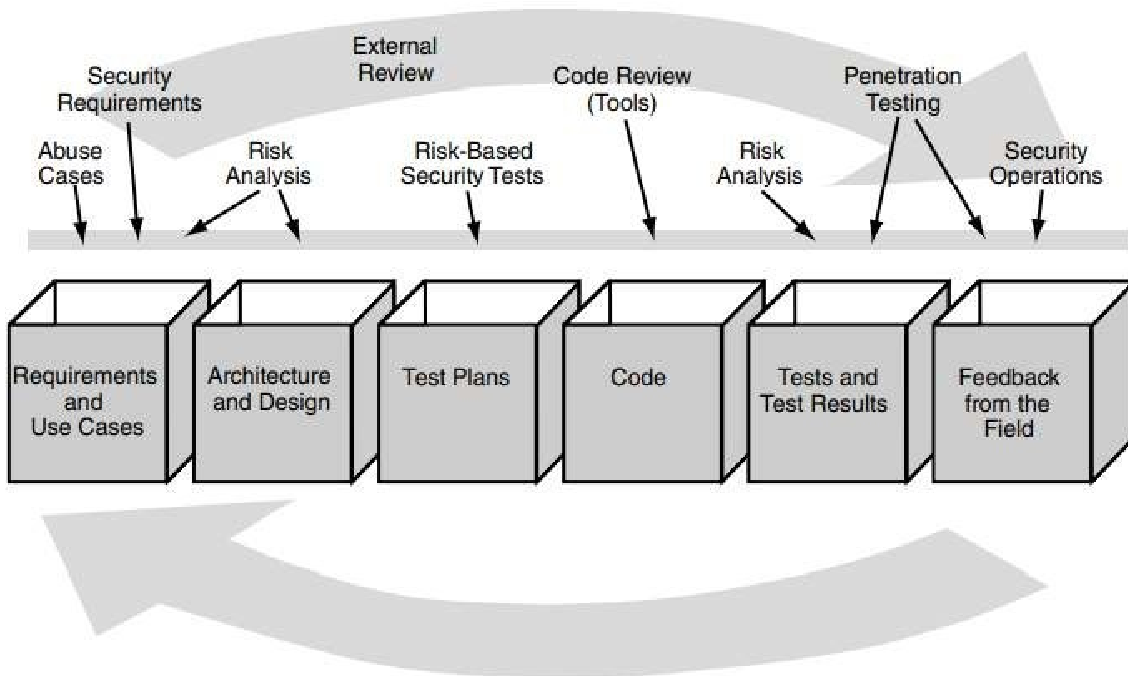


Figure : *Software development life cycle with defined security touchpoints [McGraw 2006]*

- The above Figure depicts one example of how to incorporate security into the SDLC using the concept of touchpoints. Software security best practices (touchpoints shown as arrows) are applied to a set of software artifacts (the boxes) that are created during the software development process.
- The intent of this particular approach is that it is process neutral and, therefore, can be used with a wide range of software development processes (e.g., waterfall, agile, spiral, Capability Maturity Model Integration [CMMI]).
- Security controls in the software's life cycle should not be limited to the requirements, design, code, and test phases. It is important to continue performing code reviews, security tests, strict configuration control, and quality assurance during deployment and operations to ensure that updates and patches do not add security weaknesses or malicious logic to production software.
- Additional considerations for project managers, including the effect of

software security requirements on project scope, project plans, estimating resources, and product and process measures.

DEFINING PROPERTIES OF SECURE SOFTWARE

- 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.

Core Properties of Secure Software

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

Figure : *Core security properties of secure software*

- **Confidentiality.** The software must ensure that any of its characteristics (including its **Confidentiality.** 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.

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.

<u>Influential Properties of Secure Software</u>

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

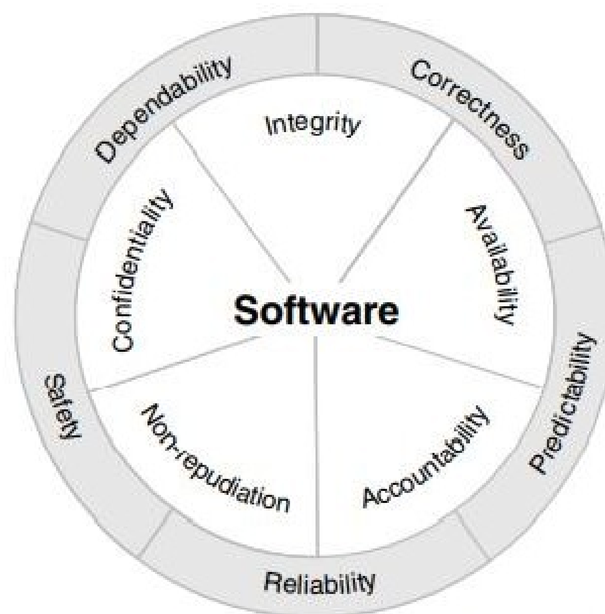


Figure : *Influential properties of secure software*

- Dependability
- Correctness
- Predictability
- Reliability
- Safety

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.
- 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.
- 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.

"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.

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

- 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.

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.
- 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.

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 touchpoints integrated throughout the SDLC; all of this must then be deployed under an umbrella of applied risk management.
- 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.
- **Touchpoints** are lightweight software security best practice activities that are applied to various software artifacts.

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 below Figure, 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.
- 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

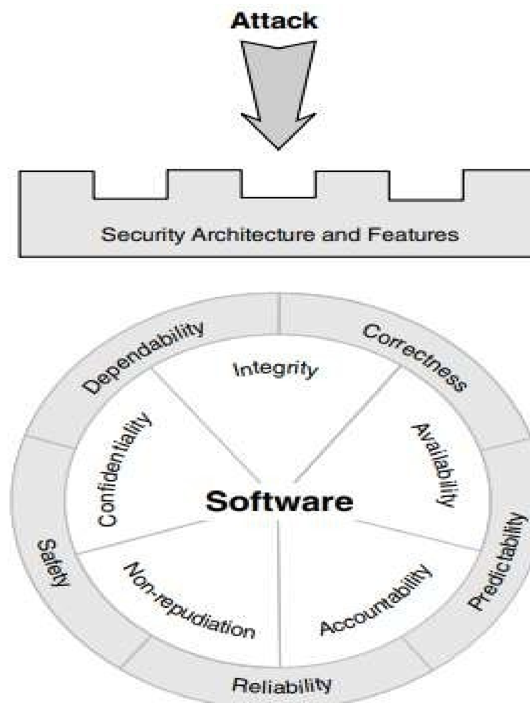


Figure : *Addressing expected issues with security architecture and features*

- **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 in below Figure), which in many cases is mislabeled as application security.

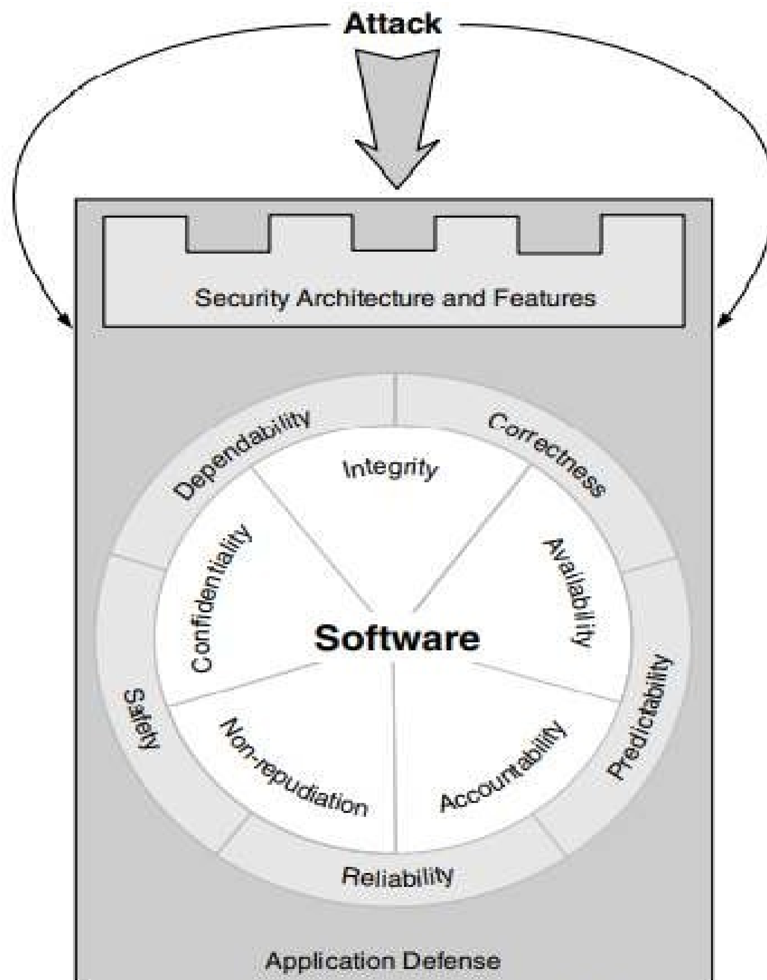


Figure : *Addressing the unexpected through application defense*

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.

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 in below Figure).

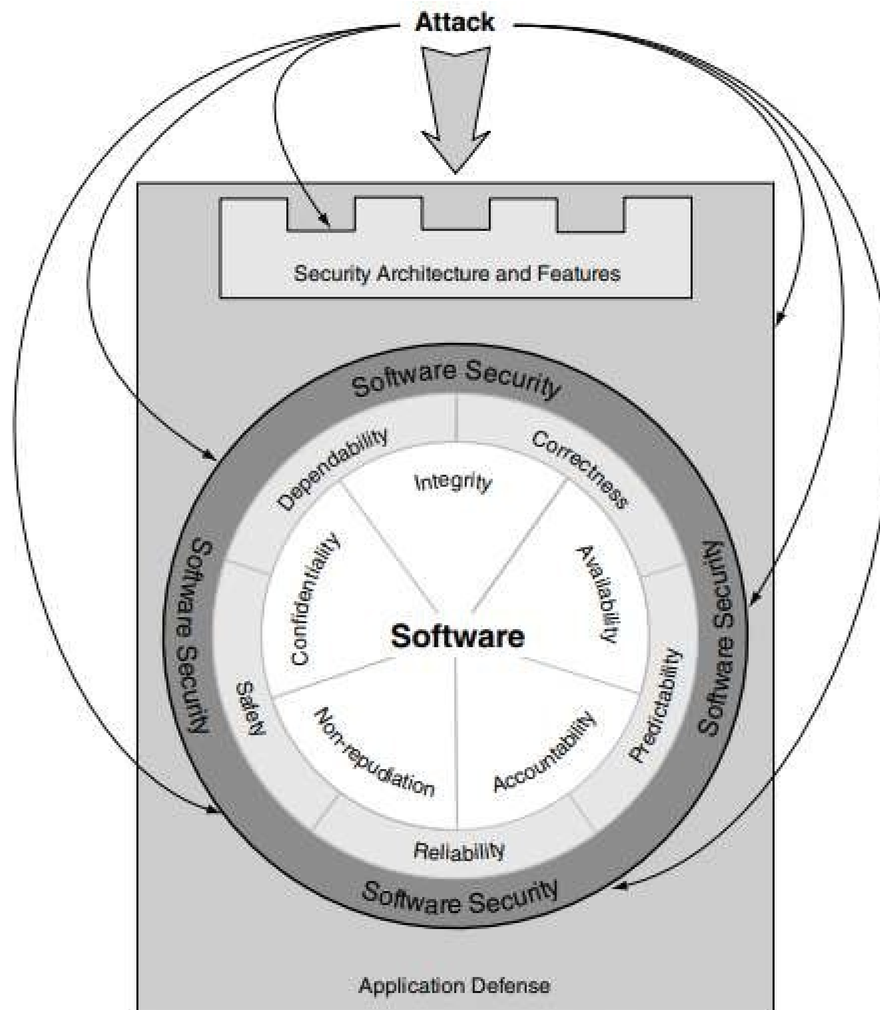


Figure : *Addressing the unexpected through software security*

- 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.

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.
- 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.

The Attacker's Perspective

- 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 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.

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 below Table.

Table: 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 number. Specific weaknesses (underlying issues that may cause vulnerabilities) should reference industrystandard identifiers such as the Common Weakness Enumeration (CWE)

Method of attack	What is the vector of attack used (e.g., malicious data entry, maliciously crafted file, protocol corruption)?
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 below Table.

Table: 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. Clientside 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."

How to Assert and Specify Desired Security Properties

- 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**.

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 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.
- 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

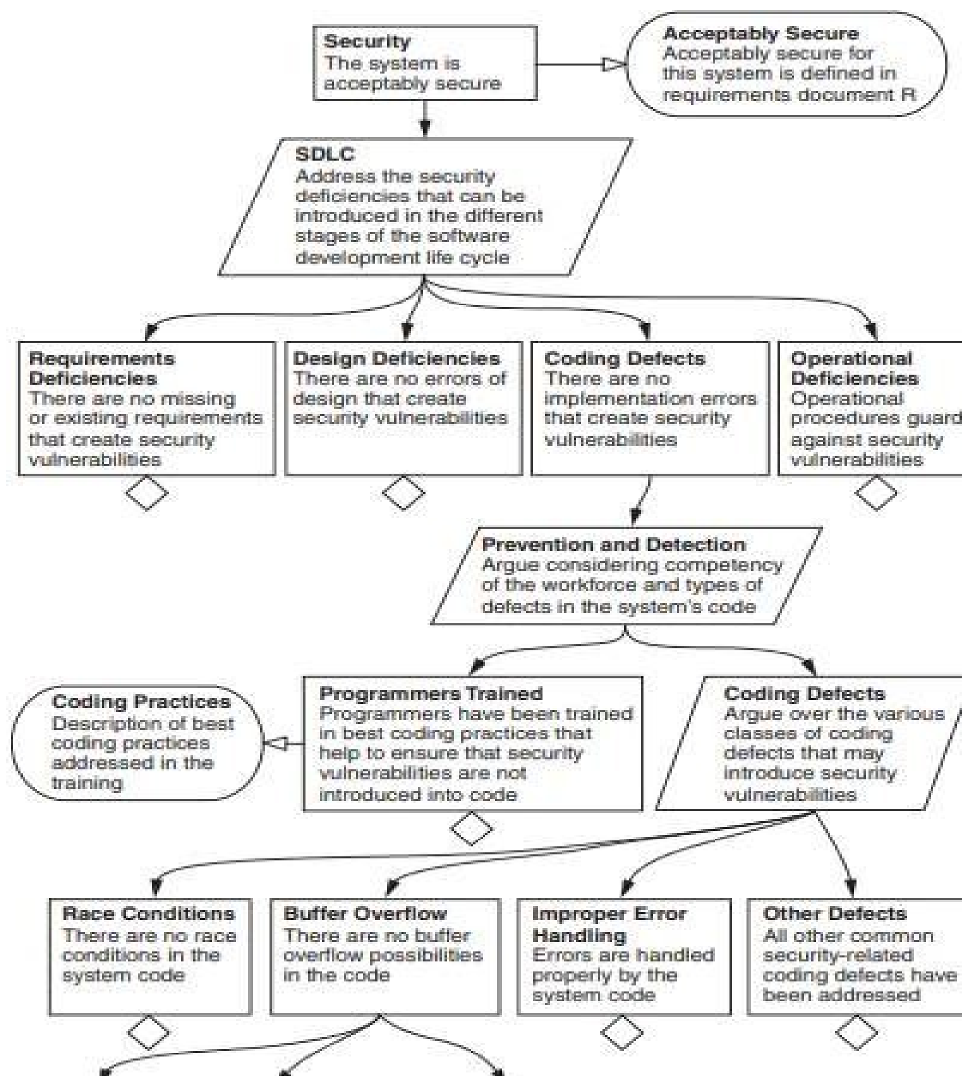
(4) the system and its components were tested with invalid arguments and all such inputs were rejected or properly handled as exceptions.

A Security Assurance Case Example

- The structure for a partially developed security assurance case focusing on buffer overflow coding defects appears in below Figure. This case is presented in a graphical notation called Goal Structuring Notation (GSN).
- 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.**
- 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.
- 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.
- In below Figure, 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.



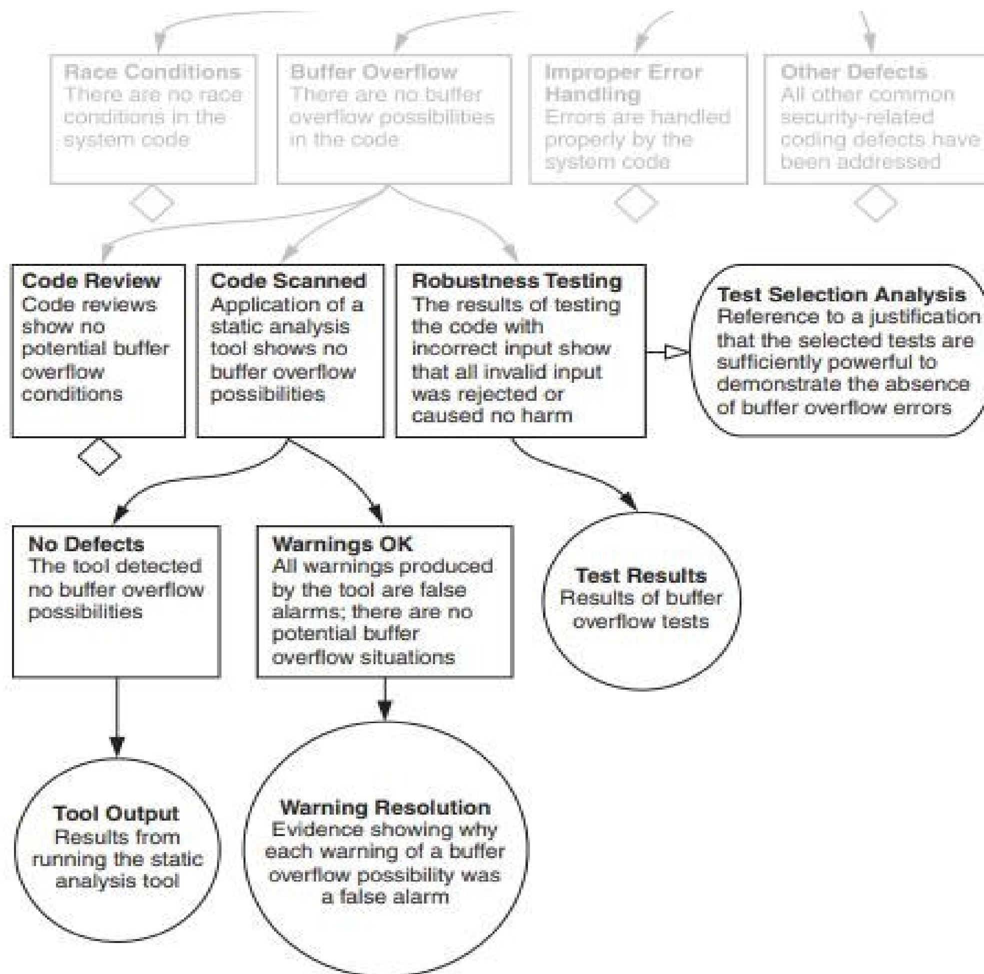


Figure : *Partially expanded security assurance case that focuses on buffer overflow*