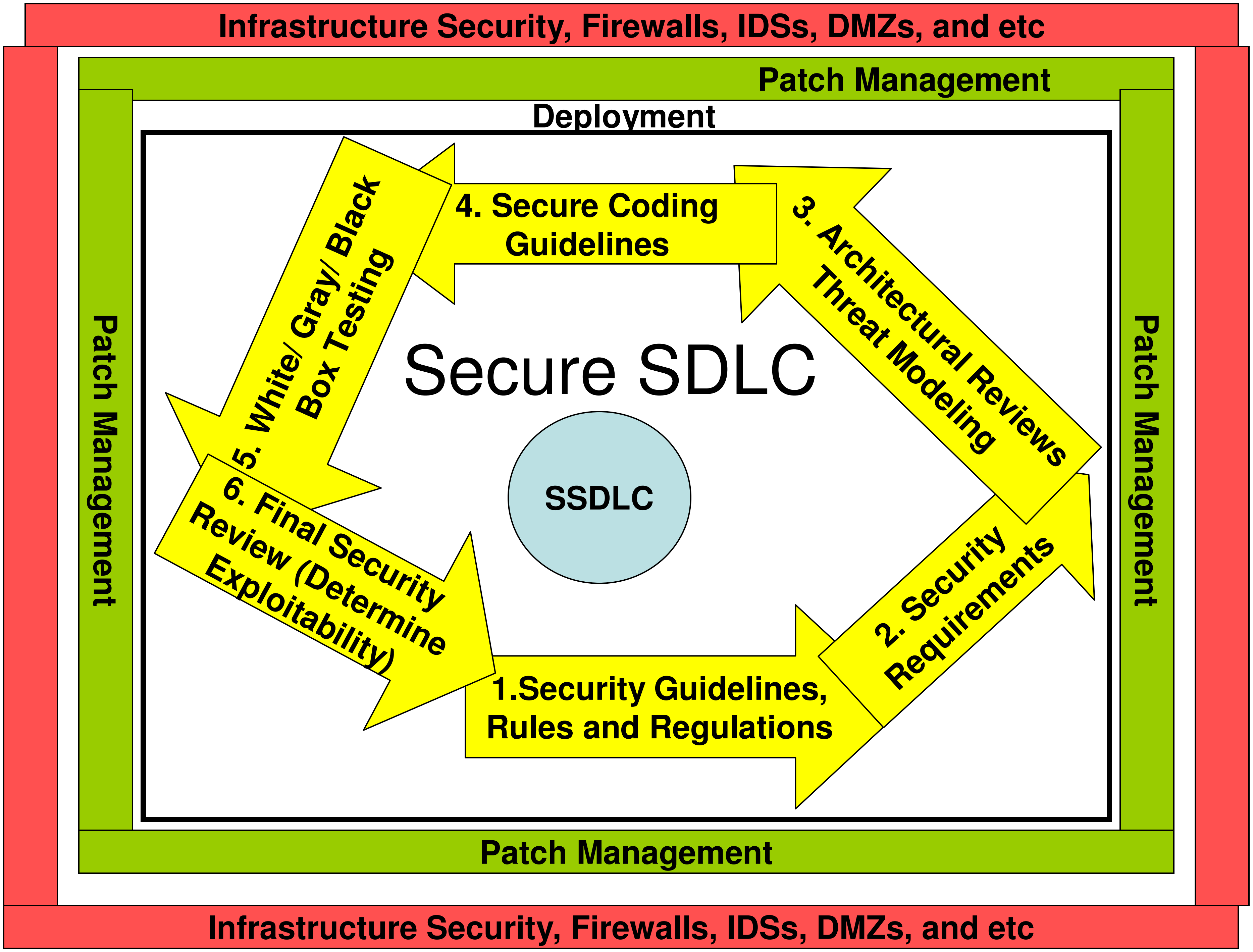
Secure Software Development Lifecycle

Dr. Natarajan Meghanathan
Associate Professor of Computer Science
Jackson State University
E-mail: natarajan.meghanathan@jsums.edu


Infrastructure Security, Firewalls, IDSs, DMZs, and etc
Patch Management
Deployment
4. Secure Coding Guidelines
3. Architectural Reviews Threat Modeling
5. White/ Gray/ Black Box Testing
Secure SDLC
SSDLC
6. Final Security Review (Determine Exploitability)
2. Security Requirements
1.Security Guidelines, Rules and Regulations
Patch Management
Patch Management
Patch Management
Infrastructure Security, Firewalls, IDSs, DMZs, and etc

Phase 1: Security Guidelines, Rules and Regulations

- Become aware of the security guidelines, models, rules and regulations (collectively called sometimes as security policies) to be adopted.
  - Even though some security experts can be hired for software development, the entire development process has to be still conducted by professionals who are not much exposed to security-related design and development in their career. So, this phase is essential at least for the time being in a secure software lifecycle.

Phase 2: Security Requirements

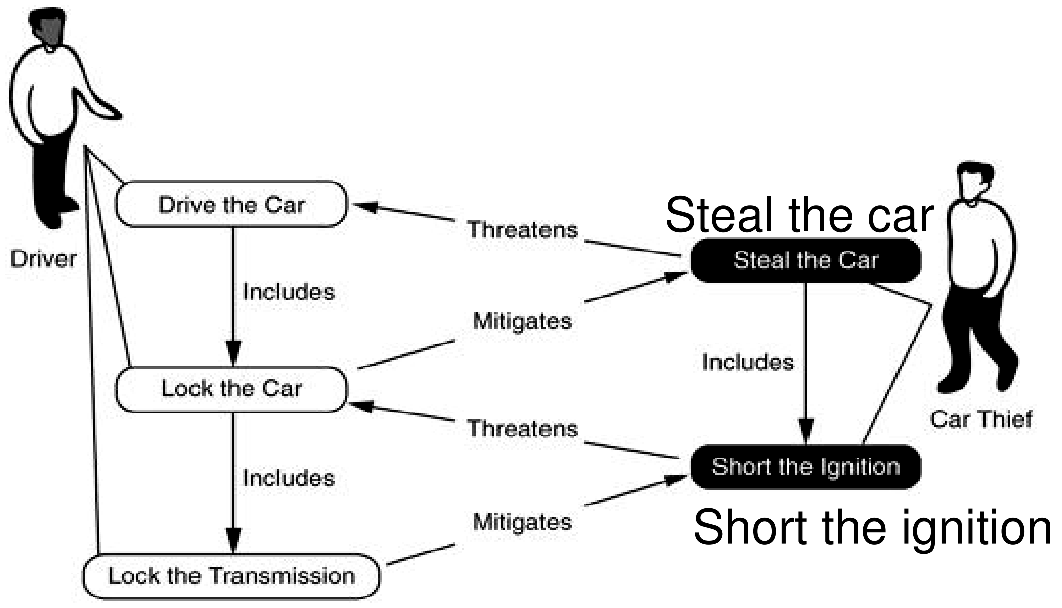
- Each functional requirement description for the software should contain a section titled: “Security Requirements” documenting any security-specific needs of that particular requirement and without incorporating it, the software will deviate from the system-wide security policy or specification.
- Misuse cases are one of the most preferred ways to document security requirements in conjunction with the use cases documenting the functional requirements.
- Security defect prevention during the requirements phase will help to detect and avoid security errors before they propagate to the later phases of the SDLC. It is cheaper to fix the defect in the requirements phase.
- Requirements traceability should be documented for each security requirement so that they can be associated with all parts of the system where it is used; any impact of changes in the security requirement can be later traced back to these parts and appropriate adjustments could be made.
- The security requirements and misuse cases later evolve to test cases to evaluate the software with respect to security.

Use Case and Misuse Case

- A use case describes the system's behavior as a response to a request originating from (an actor) outside of the system in consideration.
- Misuse case – use case from a hacker's point of view
- A misuse case diagram is created together with a corresponding use case diagram. The model introduces two new important entities (in addition to those from the traditional use case model):
  - Misuse case – a sequence of actions that can be performed by any person or entity in order to harm the system
  - Misuser – the actor that initiates the misuse case. This can either be done intentionally or inadvertently.
- The misuse case model makes use of those relation types found in the use case model: *include*, *extend*, *generalize*, and *association*. In addition, it introduces two new relations to be used in the diagram:
  - Mitigates – A use case can mitigate the chance that a misuse case will complete successfully.
  - Threatens – A misuse case can threaten a use case by exploiting it or hindering it.

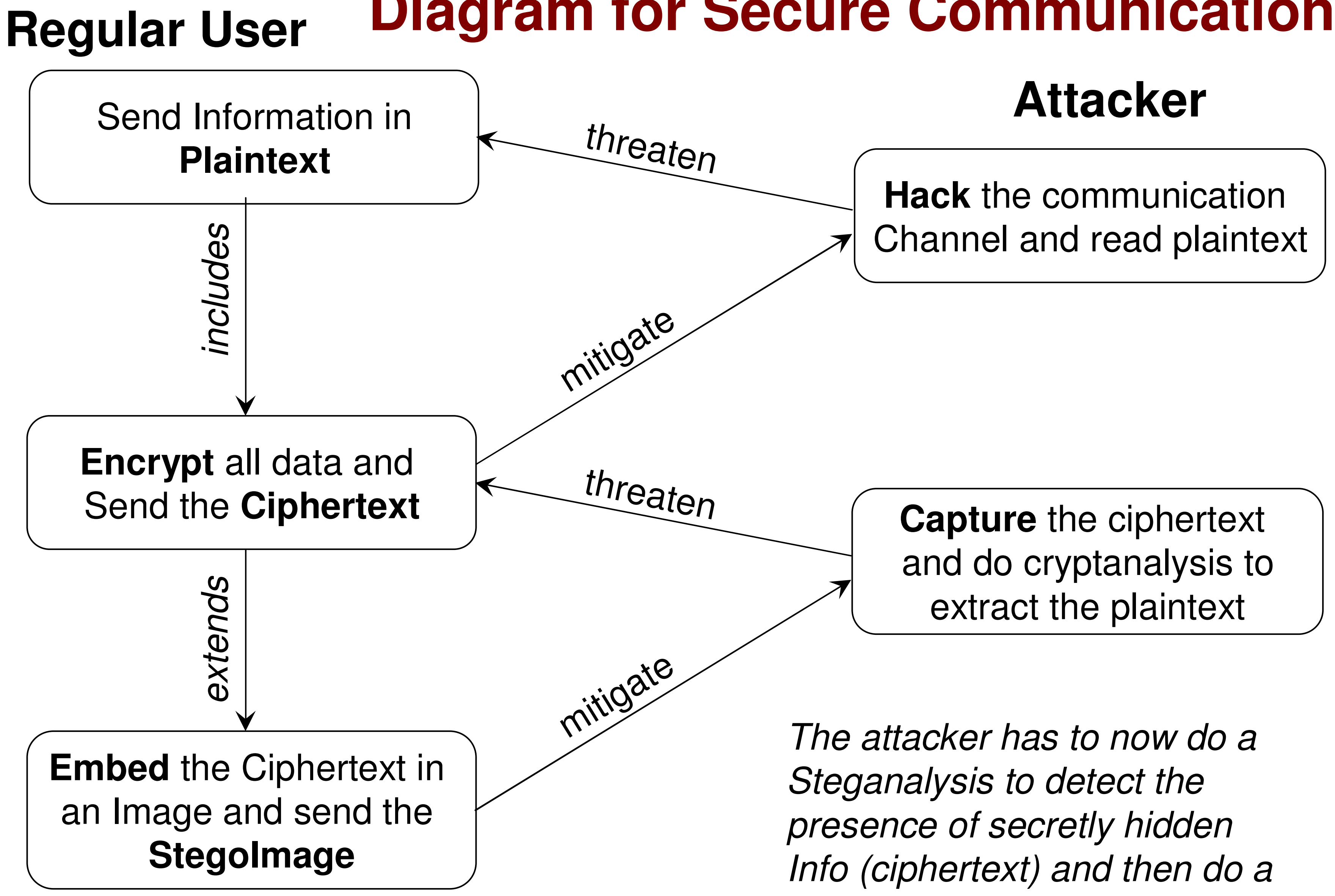
Misuse Case

- A misuse case describes scenarios of malicious acts against a system.
- A misuse case leads to the identification of negative scenarios or threats posted to the system, often leading to new requirements, which again may be expressed in new use cases to mitigate the misuse cases and so on.
  - Creating misuse cases will often trigger a chain reaction that makes it easier to identify both functional and non-functional security requirements.



Source: M. Imran Daud, “Secure Software Development Model: A Guide for Secure Software Life Cycle,” *Proceedings of the International Multi Conference on Engineers and Computer Scientists*, vol. I, March 2010.

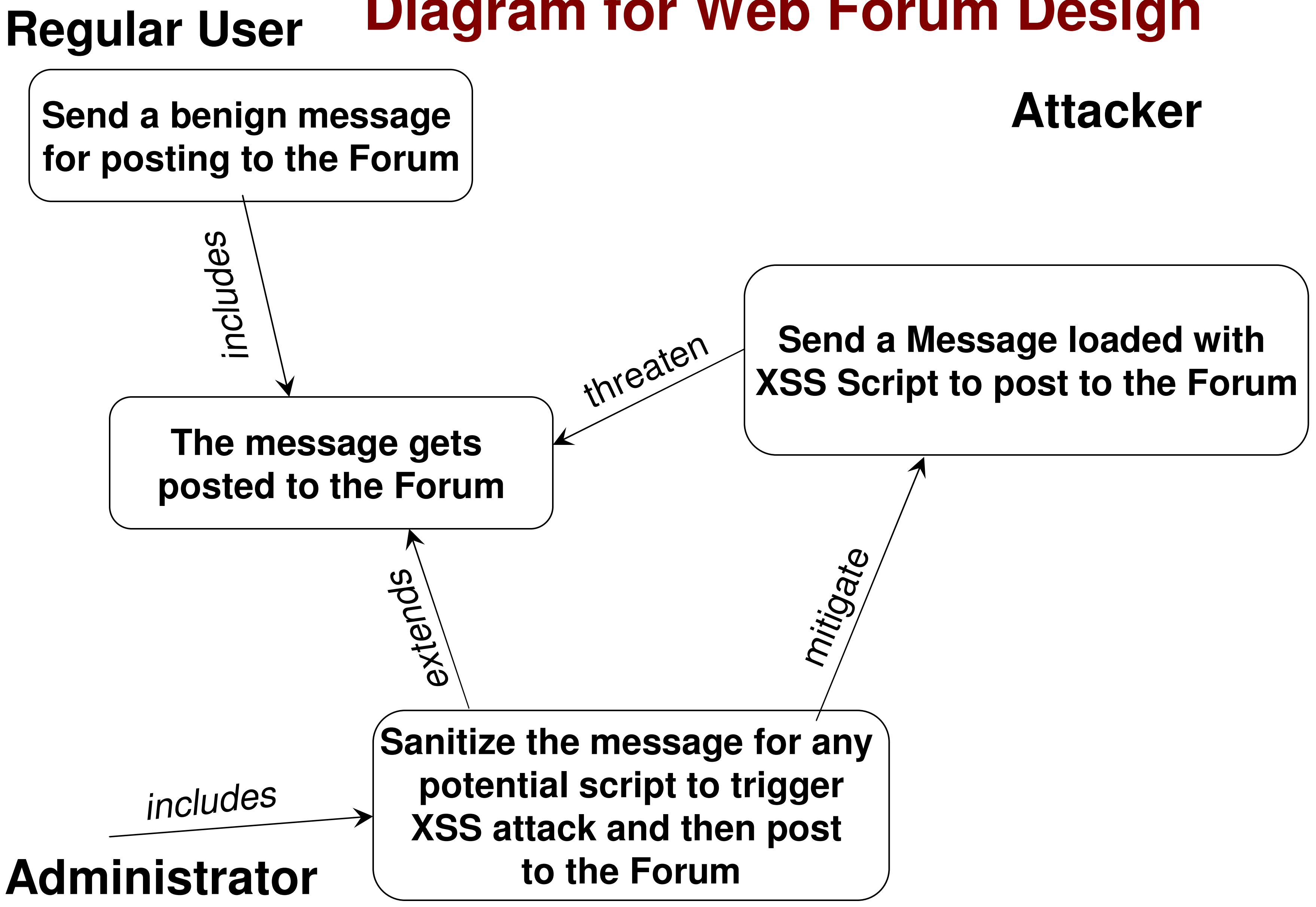
Example 1: Use Case – Misuse Case Diagram for Secure Communication



Regular User
Attacker
Send Information in **Plaintext**
Encrypt all data and Send the **Ciphertext**
Embed the Ciphertext in an Image and send the **StegoImage**
Hack the communication Channel and read plaintext
Capture the ciphertext and do cryptanalysis to extract the plaintext
includes
extends
threaten
mitigate
threaten
mitigate

The attacker has to now do a Steganalysis to detect the presence of secretly hidden Info (ciphertext) and then do a Cryptanalysis on the extracted ciphertext to extract the plaintext

Example 2: Use Case – Misuse Case Diagram for Web Forum Design


Regular User
Send a benign message for posting to the Forum
includes
The message gets posted to the Forum
Attacker
Send a Message loaded with XSS Script to post to the Forum
threaten
extends
mitigate
includes
Administrator
Sanitize the message for any potential script to trigger XSS attack and then post to the Forum

Sample Security Requirements

- Scenario 1: Application stores sensitive information that must be protected for HIPAA compliance
- Sec. Req: Strong encryption must be used to protect sensitive information wherever stored.

- Scenario 2: The application transmits sensitive user information across potentially untrusted or unsecured networks
- Sec. Req: The communication channels must incorporate encryption to prevent snooping (to protect the confidentiality of the data) and mutual cryptographic authentication must be employed to prevent man-in-the-middle attacks (for integrity and authenticity of communication)

- Scenario 3: The application must remain available to legitimate users.
- Sec. Req: Resource utilization by remote users must be monitored and limited to prevent or mitigate denial-of-service attacks.

Sample Security Requirements

- Scenario 4: The application supports multiple users with different levels of privilege.
- Sec. Req: The application should define the actions that users at each privilege level is authorized to perform. The various privilege levels assigned to users should be tested. Mitigations for authorization bypass attacks need to be defined.

- Scenario 5: The application takes user input and uses SQL.
- Sec. Req: SQL injection mitigations need to be defined.

- Scenario 6: The application manages sessions for a logged-in user.
- Sec. Req: Session hijacking mitigations should be in place.

- Scenario 7: The system needs to keep track of individual users and authentication must be enforced.
- Sec. Req: User passwords should be securely stored and mitigations to combat dictionary attacks must be in place.

Sample Security Requirements

- Scenario 8: The application is written in C or C++.
- Sec. Req: The code must be written in such a way that buffer sizes are always tracked and checked; format strings should not be modified by user input; and integer values should not be allowed to overflow. If the compiler supports the use of stack canaries, use them.

- Scenario 9: The application presents user-generated data in HTML.
- Sec. Req: Mitigations for XSS attacks must be in place.

- Scenario 10: The application requires an audit log.
- Sec. Req: Define all functions that need to be logged; Verify that the audit log is secure.

- Scenario 11: The application uses cryptography
- Sec. Req: The generated secrets must use a secure random-number generator.

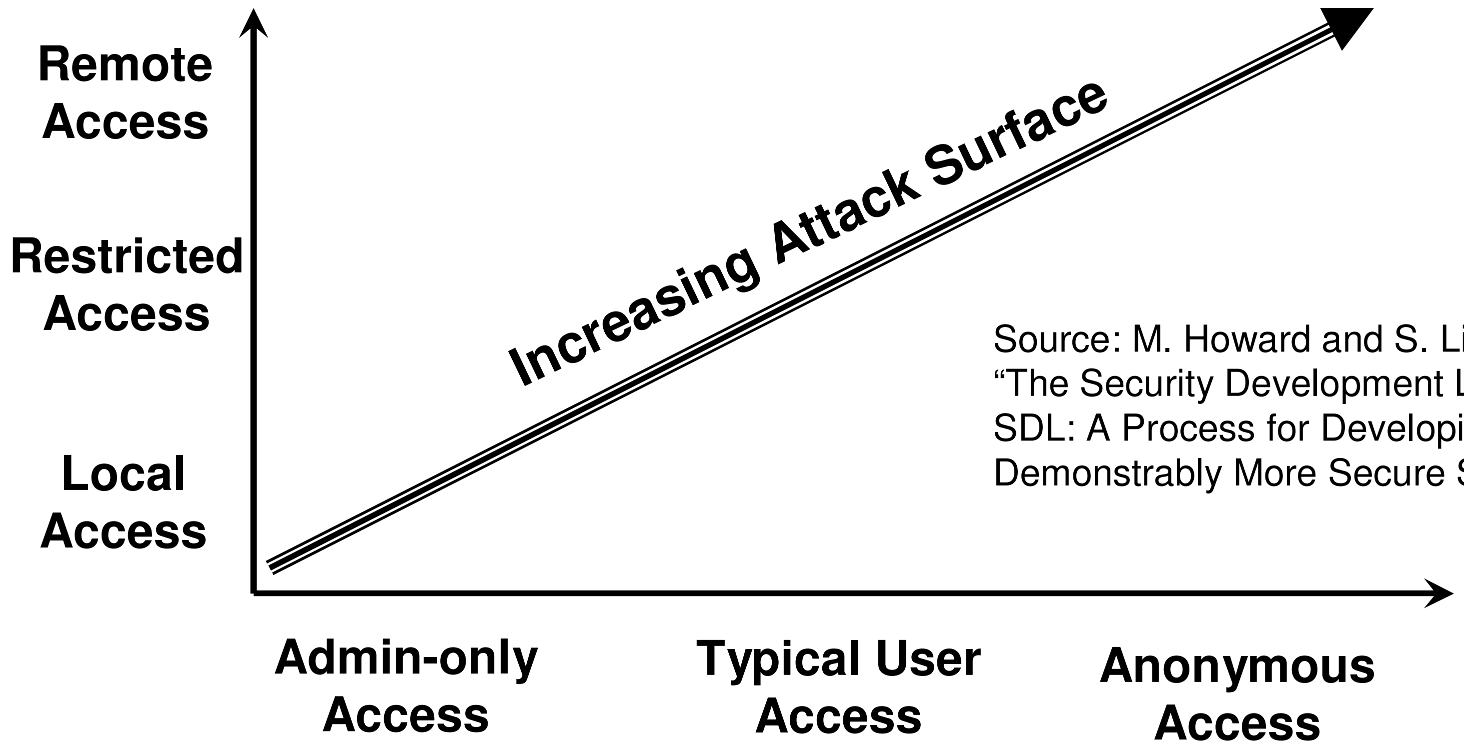
- Scenario 12: The application opens files that are typically exchanged over untrusted links such as a media file over the Internet.
- Sec. Req: The application must validate all data read from the file and not trust it.

Phase 3: Secure Software Design

- The first stage of secure software design is related to identifying the potential threats to an application and finding ways to minimize the risk of those threats
  - This is accomplished by conducting activities that approach the design from an adversary's perspective that includes identifying the pathways (attack surface) that could be used to conduct an attack.
- <u>Attack Surface Evaluation</u>
- Attack Surface – Entry and exit points of an application that are accessible to users and attackers.
- Entry points – are the inputs to the application through interfaces, services, protocols and code
- Exit points – are the outputs from the application, including error messages produced by the application, in response to user interaction.
- The entry and exit points should be accessible only to users who possess the required level of trust.
- Attack surface evaluation is aimed at analyzing and reducing the attack surface of a software application.
- A smaller attack surface mitigates security risk by making the exploitation harder and by lowering the exploitation's damage.

Accessibility Increases Attack Surface

- Requiring authentication and limiting access to entry points and exit points in the application significantly reduces the attack surface.
- For example, an entry point that is restricted to local access by an administrator has a smaller attack surface than an entry point exposed to remote access by an anonymous user.



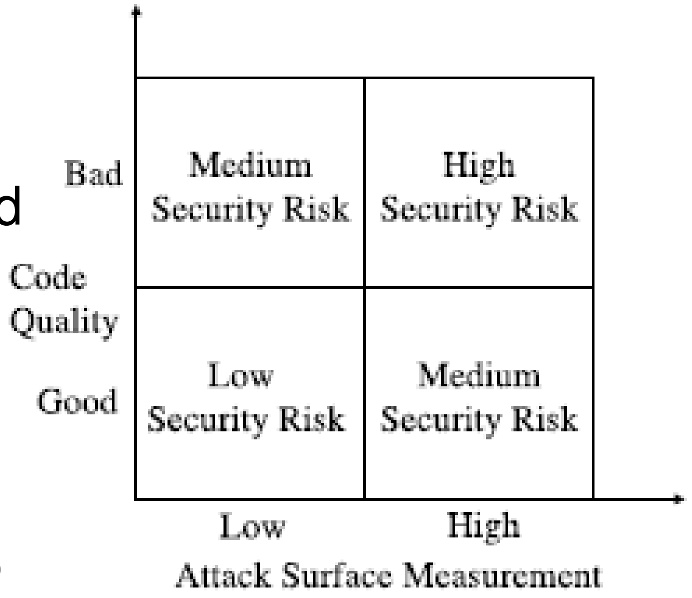
Source: M. Howard and S. Lipner, "The Security Development Lifecycle, SDL: A Process for Developing Demonstrably More Secure Software," 2006.

UDP vs. TCP: Impact on Attack Surface

- UDP has a larger attack surface than TCP
  - UDP is a connectionless protocol that is used to send a datagram to a destination without prior arrangement. Hence, it is possible to “fire and forget” multiple UDP datagrams to a destination from spoofed IP addresses.
  - TCP is a connection-oriented protocol wherein the source and destination go through a three-way handshake before the actual data transfer. Hence, once a connection is formed, it is not possible to “fire and forget” datagrams from randomly spoofed IP addresses.
  - Also, the connection establishment process of TCP can be made more secure, if the communication is done on the top of IPSec – this way, denial-of-service attacks by spoofing the connection requests can be avoided.
- Support for UDP in an application should be removed, if not needed.

Design Principles: Smaller Attack Surface

- Economy of Mechanism
  - Keep the design as simple and small as possible.
  - Complexity in the design leads to more errors in the code, affecting its quality, and the security implications get difficult to understand
  - To apply the economy of mechanism to secure software design, we need to determine the minimum amount of functionality required by the application to perform its tasks and provide only the features that implement that functionality.
  - If the software must provide a number of features to a large and diverse set of users, it should be possible for a user to turn off those features or disable them if they are not necessary.



Source: http://www.cs.cmu.edu/~pratyus/tse10.pdf

Principles of Secure Software Design

- Principles define effective practices that are applicable primarily to design architecture-level software decisions and are recommended regardless of the platform or the language of the software
  - Sometimes, the principles may exist in opposition to each other, so appropriate tradeoffs must be made, if required.
- Principles
  - Securing the weakest link
  - Defense in depth
  - Failing securely
  - Least privilege
  - Separation of privilege
  - Economy of mechanism
  - Reluctance to trust
  - Never assume that your secrets are safe
  - Complete mediation
  - Promoting privacy

Principles of Secure Software Design

- Securing the weakest link: Attackers are more likely to attack a weak spot (for e.g., servers – endpoints of communication) in a software system and follow the path of least resistance, than to penetrate a heavily fortified component (e.g., cryptographic algorithms, firewalls, etc).
  - Note: All the links are essential; attackers target the weakest link.
  - If a bad guy wants access to secret data sent from point A to point B, a clever attacker will target one of the endpoints, try to find a flaw like a buffer overflow, and then look at the data before it gets encrypted, or after it gets decrypted.

- Defense in depth: Defending an application with multiple overlapping layers can prevent a single point of failure that compromises the security of the application.
  - Note: All the links are not essential. However, presence of more than one layer of defense increases the security.
  - Use of a packet-filtering router in conjunction with an application gateway and an intrusion detection system along with good password controls (a strong authentication system) and an adequate user training increase the work of an attacker.

Principles of Secure Software Design

- Fail securely: Secure systems should have a well-defined status after failure, either to a secure failure state or via a recovery procedure to a known secure state.
  - The default configuration settings for an application should be the most secure settings possible. Such a design is called secure-by-default.
- Least Privilege: A user should be assigned only the minimum necessary rights needed to access a requested resource and these rights should be in effect for the shortest duration necessary (remember to relinquish privileges).
  - The function of the subject (as opposed to its identity) should control the assignment of rights. Role-based access control may be more appropriate.
  - If the subject does not need access to an object (even if it is entitled to) to perform its task, it should not have the right to access that object. For example, if a subject needs to append to an object, but not alter the information already contained in the object, it should be given only “append” rights and not “write” rights.
  - Reduce Privilege Escalations: Do not run a program as a member of the local administrators group, unless elevated privileges are needed. If a security vulnerability is found in the code and an attacker can inject code into the process, make the code perform sensitive tasks, or run a Trojan horse or virus, the malicious code will run with the same privileges as a compromised process. If the process is running as an administrator, the malicious code runs as an administrator.

Principles of Secure Software Design

- Separation of Privilege: A system should not grant permission based upon a single condition.
  - A protection mechanism that requires two keys (validated by two physically distinct programs) to unlock it is more robust and flexible than one that allows access to the presenter of only a single key.
- Reluctance to Trust: Developers should assume that the environment in which their system resides is insecure.
  - Unless proven to be trustworthy, all external stimuli have the potential to be an attack and assuming external systems are insecure would increase the defense in depth.
- Never assume that your secrets are safe: Always assume that an attacker knows everything (including source code and design) that you know.
  - Tools such as decompilers and disassemblers allow attackers to obtain sensitive information that may be stored in binary files.
- Do not cache access control decisions: Every access to every object must be checked for authority.
  - Operating systems typically validate user requests for a file descriptors at the time of their creation to make sure the user has the required access. However, if the owner of the object revokes the permission granted to the user, and the file descriptor is still in use, the user can continue to access the object despite the revocation.