SEZG566/SSZG566

Secure Software Engineering

Architecting/Designing for Security

T V Rao

BITS Pilani

Pilani | Dubai | Goa | Hyderabad

- *The slides presented here are obtained from the authors of the books, product documentations, and from various other contributors. I hereby acknowledge all the contributors for their material and inputs.*
- *I have added and modified slides to suit the requirements of the course.*

BITS Pilani

Pilani | Dubai | Goa | Hyderabad

Secure Architecture & Design - Introduction

Nomenclature (SWEBOK)

Software design is the activity that uses software requirements to produce a description of the software's internal structure that will serve as the basis for its construction

Software design consists of two activities :

- Software architectural design (sometimes called high-level design): develops top-level structure and organization of the software and identifies the various components.
- Software detailed design: specifies each component in sufficient detail to facilitate its construction.

Understanding Software Architecture

The software architecture of a program or computing system is the structure or structures of the system which comprise

– The software components

– The externally visible properties of those components

– The relationships among the components

Software architectural design represents the structure of the data and program components that are required to build a computer-based system

An architectural design model is transferable

– It can be applied to the design of other systems

– It represents a set of abstractions that enable software engineers to describe architecture in predictable ways

Importance of Software Architecture

- Representations of software architecture are an enabler for communication between all stakeholders interested in the development of a computer-based system
- The software architecture highlights early design decisions that will have a profound impact on all software engineering work that follows and, as important, on the ultimate success of the system as an operational entity
- The software architecture constitutes a relatively small, intellectually graspable model of how the system is structured and how its components work together

Uses of software architecture descriptions

- **Reuse**: Architecture descriptions can help software reuse.
  - The software engineering world has, for a long time, been working towards a discipline where software can be assembled from parts that are developed by different people and are available for others to use.
- **Construction and Evolution**: As architecture partitions the system into parts, some architecture provided partitioning can naturally be used for constructing the system
  - which also requires that the system be broken into parts such that different teams (or individuals) can separately work on different parts.
- **Analysis**: It is highly desirable if some important properties about the behaviour of the system can be determined before the system is actually built.
  - This will allow the designers to consider alternatives and select the one that will best suit the needs.

General Objectives of Software Architecture and Design

Completeness

- Supports the full scope of the defined requirements

Stability

- Consistently performs as intended within its defined operational context

Flexibility

- Can adapt to changing conditions
- Decompose such that selected components can be replaced going forward with minimal impact to the software

Extensibility

- Leverages industry standards
- Long-lived and resistant to obsolescence

Scalability

- Operates effectively at any size and load

Security-Specific Objectives of Software Architecture and Design

Comprehensive functional security architecture

- Security features and capabilities are fully enabled

Attack resistance

- Contains minimal security weaknesses that could be exploited

Attack tolerance

- While resisting attack, software function and capability are not unduly affected

Attack resilience

- In the face of successful attack, the effects on the software are minimized
- Operates effectively at any size and load

How to Design

A designer must practice diversification and convergence -[Belady]

- The designer selects from design components, component solutions, and knowledge available through catalogs, textbooks, and experience
- The designer then chooses the elements from this collection that meet the requirements defined by requirements engineering and analysis modeling
- Convergence occurs as alternatives are considered and rejected until one particular configuration of components is chosen

Software design is an iterative process

- As design iteration occurs, refinements lead to design representations at lower levels of abstraction

BITS Pilani

Pilani | Dubai | Goa | Hyderabad

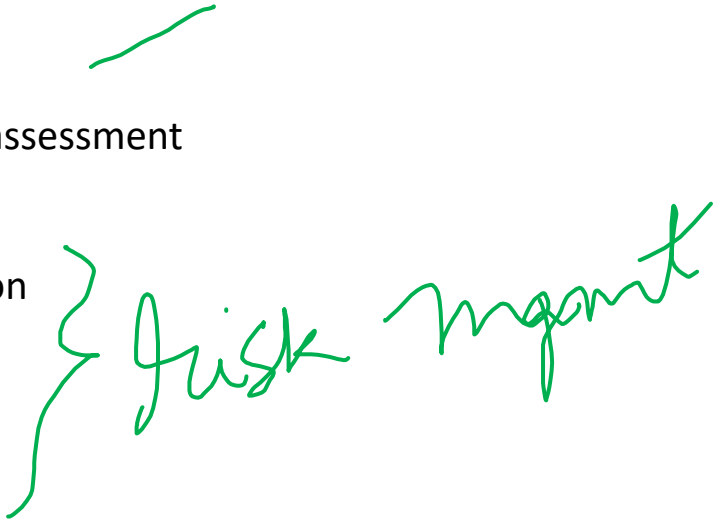
Secure Architecture Risk Analysis

Architectural Issues

- Security architecture (the architecture of security components, e.g. firewall, encryption mechanism) is not same as secure architecture (i.e. resilient and resistant to attacks)
- Secure architecture not only must address known weaknesses and attacks, but must be flexible and resilient under changing security conditions
- Architects (and designers) must focus on minimizing the risk profile. It requires complex and diverse knowledge (both on threats and technologies).

Architectural Risk Analysis

- The risk assessment methodology encompasses six fundamental activity stages:
  - software characterization
  - architectural vulnerability assessment
  - threat analysis
  - risk likelihood determination
  - risk impact determination
  - risk mitigation



Software Characterization

- Assessing the architectural risks for a software system is easier when the boundaries of the software system are identified, along with the resources, integration points, and information that constitute the system
- The following artifacts may be used for review:

- software business case
- functional and non-functional requirements
- enterprise architecture requirements
- use case documents
- misuse and abuse case documents
- software architecture documents describing logical, physical, and process views
- data architecture documents
- detailed design documents such as UML diagrams that show behavioral and structural aspects of the system
- software development plan
- transactions
- security architecture documents
- identity services and management architecture documents
- quality assurance plan
- test plan/acceptance plan
- risk list / risk management plan
- problem resolution plan
- issues list
- project metrics
- programming guidelines
- configuration and change management plan
- project management plan
- disaster recovery plan
- system logs
- operational guides

Idea is to get comprehensive, yet concise picture of the nature of application

Architectural Risk Analysis

- Architectural risk analysis examines the preconditions that must be present for vulnerabilities to be exploited and assesses the states that the system may enter upon exploitation.
  - assess vulnerabilities not just at a component or function level, but also at interaction points
  - risk analysis testing can only prove the presence, not the absence, of flaws

- Three activities can guide architectural risk analysis:
  - known vulnerability analysis,
  - ambiguity analysis, and
  - underlying platform vulnerability analysis

Known Vulnerability Analysis

- Consider the architecture against a body of known bad practices or known good principles for confidentiality, integrity, and availability
  - e.g., the good principle of "*least privilege*" prescribes that all software operations should be performed with the least possible privilege.
  - Diagram the system's major modules, classes, or subsystems and circle areas of high privilege versus areas of low privilege. Consider the boundaries between these areas and the kinds of communications across those boundaries.

Mitre CWE

Ambiguity Analysis

- Ambiguity can be a source of vulnerabilities when it exists between requirements or specifications and development.
  - Note places where the requirements are ambiguously stated and the implementation and architecture either disagree or fail to resolve the ambiguity.
    - e.g. , a requirement for a web application might state that an administrator can lock an account and the user can no longer log in while the account remains locked. What about sessions for that user that are actively in use at the time the administrator locks the account? Is the user suddenly and forcibly logged out, or is the active session still valid until the user logs out?

Underlying Platform Vulnerability Analysis

- Carry out analysis of the vulnerabilities associated with the application's execution environment including operating system vulnerabilities, network vulnerabilities, platform vulnerabilities, and interaction vulnerabilities resulting from the interaction of components.
- There are web sites that aggregate vulnerability information. These sites and lists should be consulted regularly to keep the vulnerability list current for a given architecture.

BITS Pilani

Pilani | Dubai | Goa | Hyderabad

Principles for Secure Design

Fundamental Design Concepts

Abstraction—data, procedure, control

Patterns—"conveys the essence" of a proven design solution

Separation of concerns—any complex problem can be more easily handled if it is subdivided into pieces

Modularity—compartmentalization of data and function

Information Hiding—controlled interfaces

Functional independence—single-minded function and low coupling

Refinement—elaboration of detail for all abstractions

Aspects—a mechanism for understanding how global requirements affect design

Refactoring—a reorganization technique that simplifies the design

The beginning of wisdom (for a software engineer) is to recognize the difference between getting program to work, and getting it right – *M A Jackson*

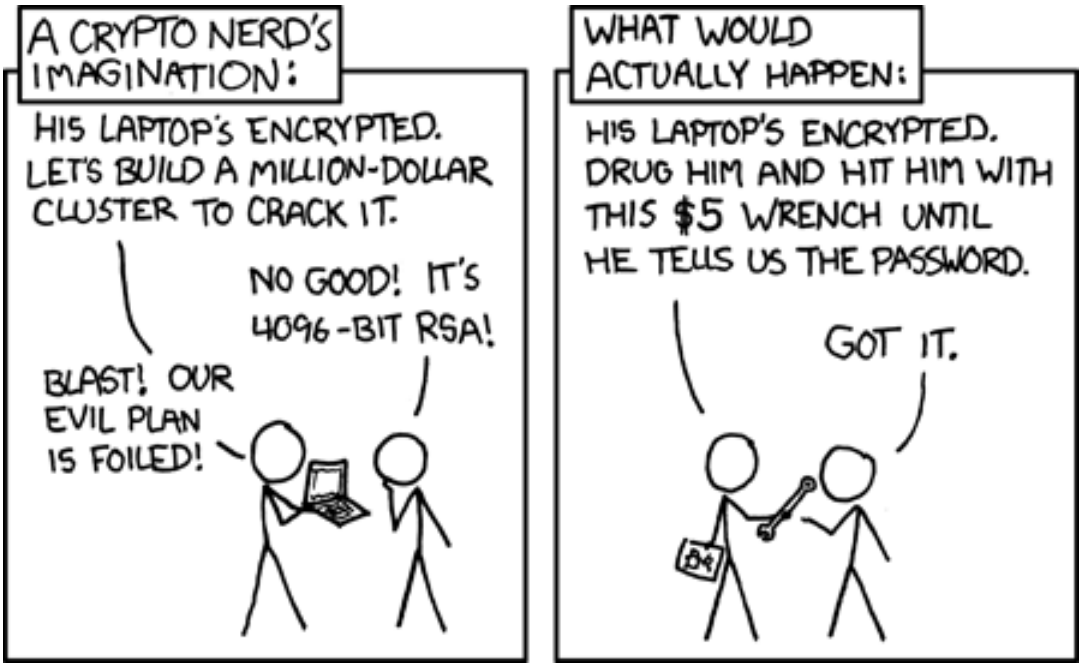
Design Principles for Software Security

- Securing the Weakest Link
- Defense in Depth
- Failing Securely
- Least Privilege
- Separation of Privilege
- Economy of Mechanism
- Least Common Mechanism
- Reluctance to Trust
- Never Assuming that your Secrets are Safe
- Complete Mediation
- Psychological Acceptability
- Promoting Privacy

Securing the Weakest Link

- A software security system is only as secure as its weakest component
- Some cryptographic algorithms can take many years to break, but the endpoints of communication (e.g., servers) may be much easier to attack.
- Attackers don't attack a firewall unless there's a well-known vulnerability in the firewall itself (something all too common, unfortunately). they'll try to break the applications that are visible through the firewall, since these applications tend to be much easier targets
- Sometimes it's not the software that is the weakest link in your system; e.g., consider social engineering, an attack in which a bad guy uses social manipulation to break into a system

Securing the Weakest Link



https://xkcd.com/538/

Defense in depth

- Layered security mechanisms increase security of the system as a whole
- If an attack causes one security mechanism to fail, other mechanisms may still provide the necessary security to protect the system
- A software system with authentication checks may prevent intrusion by subverting a firewall.
- Implementing a defense-in-depth strategy may add to the complexity of an application, that might bring new risks with it
  - e.g., increasing the required password length from eight characters to 15 characters may result in users writing their passwords down, thus decreasing the overall security to the system
  - however, adding a smart-card requirement to authenticate to the application would add a complementary layer to the authentication process & can be beneficial.

Defense in depth

innovate achieve lead

Success for an attack involves 5 key milestones

- First, the attacker must deliver their attack.
- Second, this attack must exploit a vulnerability (person, software).
- Next, that exploit enables the installation of the attacker's bad software.
- This installation allows the command and control centers of the attackers to connect to the compromised environment.
- At this point, the final state of the exploitation can occur where the ultimate goals of the attacker have been met.

Implementing a single mechanism of defense for each key milestone will provide depth of coverage.

Defense in depth is a structuring of IT security to slow or stop any given attack with multiple mechanisms across the **cyber kill chain.**

https://seic.com/knowledge-center/field-landmines-layered-resiliency-security

Failing Securely

When a system fails, it should do so securely.

– e.g. on failure undo changes and restore to a secure state; always check return values for failure; and in conditional code/filters make sure that there is a default case that does the right thing. The confidentiality and integrity of a system should remain even though availability has been lost.

```
DWORD dwRet = IsAccessAllowed(...);
if (dwRet == ERROR_ACCESS_DENIED)
{
  // Security check failed.
  // Inform user that access is denied.
} else {
  // Security check OK.
}
```

```
DWORD dwRet = IsAccessAllowed(...);
if (dwRet == NO_ERROR) {
  // Secure check OK.
  // Perform task.
} else {
  // Security check failed.
  // Inform user that access is denied.
}
```

Least Privilege & Separation of Privilege

- Only the minimum necessary rights should be assigned to a subject that requests access to a resource and should be in effect for the shortest duration necessary (remember to relinquish privileges).
- According to Saltzer and Schroeder [Saltzer 75] in "Basic Principles of Information Protection," Every program and every user of the system should operate using the least set of privileges necessary to complete the job. if a question arises related to misuse of a privilege, the number of things that must be audited is minimized.
  - a programmer who may need to read some sort of data object, but assigns higher privilege, since "Someday I might need to write to this object, and it would suck to have to go back and change this request."
- A system should ensure that multiple conditions are met before granting permissions to an object. If an attacker is able to obtain one privilege but not a second, he or she may not be able to launch a successful attack.
- a protection mechanism that requires two keys to unlock it is more robust and flexible than one that allows access to the presenter of only a single key
  - This principle is often used in bank safe-deposit boxes. It is also at work in the defense system that fires a nuclear weapon only if two different people both give the correct command.

Economy of Mechanism

- If the design, implementation, or security mechanisms are highly complex, then the likelihood of security vulnerabilities increases. Subtle problems in complex systems may be difficult to find, especially in copious amounts of code.
- Simplifying design or code is not always easy, but developers should strive for implementing simpler systems when possible.
- The checking and testing process is less complex, because fewer components and cases need to be tested.
- Complex mechanisms often make assumptions about the system and environment in which they run. If these assumptions are incorrect, security problems may result.

Least Common Mechanism

- Avoid having multiple subjects sharing mechanisms to grant access to a resource. For e.g., serving an application on the Internet allows both attackers and users to gain access to the application.
- Every shared mechanism (especially one involving shared variables) represents a potential information path between users and must be designed with great care to be sure it does not unintentionally compromise security.
- Example : A web site provides electronic commerce services for a major company. Attackers flood the site with messages, and tie up the electronic commerce services. Legitimate customers are unable to access the web site and, as a result, take their business elsewhere.
  - Here, the sharing of the Internet with the attackers' sites caused the attack to succeed. The appropriate countermeasure would be include proxy servers or traffic throttling. The former targets suspect connections; the latter reduces load on the relevant segment of the network indiscriminately.

Reluctance to Trust

- Developers should assume that the environment in which their system resides is insecure
- Trust in external systems, code, people, etc., should always be closely held and never loosely given.
- Software engineers should anticipate malformed input from unknown users
- Users are susceptible to social engineering attacks, making them potential threats to a system
- No system is one hundred percent secure, so the interface between two systems should be secured.

Reluctance to Trust

- Point to remember is that trust is transitive. Once you dole out some trust, you often implicitly extend it to anyone the trusted entity may trust
- Hiding secrets in client code is risky. talented end users will be able to abuse the client and steal all its secrets
- According to Viega and McGraw, there are hundreds of products from security vendors with gaping security holes; Many security products introduce more risk than they address
  - Beware of vendors who resort to technobabble using newly invented terms or trademarked terms without actually explaining how the system works
  - Avoid software which uses secret algorithms. ``hackers'' can reverse-engineer the program to see how it works anyway
- According to Bishop, an entity is trustworthy if there is sufficient credible evidence leading one to believe that the system will meet a set of given requirements. Trust is a measure of trustworthiness, relying on the evidence provided. These definitions emphasize that calling something "trusted" or "trustworthy" does not make it so

Never Assuming That Your Secrets Are Safe

Relying on an obscure design or implementation does not guarantee that a system is secured. You should always assume that an attacker can obtain enough information about your system to launch an attack.

- Tools such as decompilers and disassemblers allow attackers to obtain sensitive information that may be stored in binary files.
- According to Viega and McGraw, for years, there was an arms race and an associated escalation in techniques of vendors and hackers; vendors would try harder to keep people from finding the secrets to "unlock" software, and the software crackers would try harder to break the software. For the most part, the crackers won.
- According to Viega and McGraw, the most common threat to companies is the insider attack; but many companies say "That won't happen to us; we trust our employees." The infamous FBI spy Richard P. Hanssen carried out the ultimate insider attack against U.S. classified networks for over 15 years.

Complete Mediation

A software system that requires access checks to an object each time a subject requests access, especially for security-critical objects, decreases the chances of mistakenly giving elevated permissions to that subject.

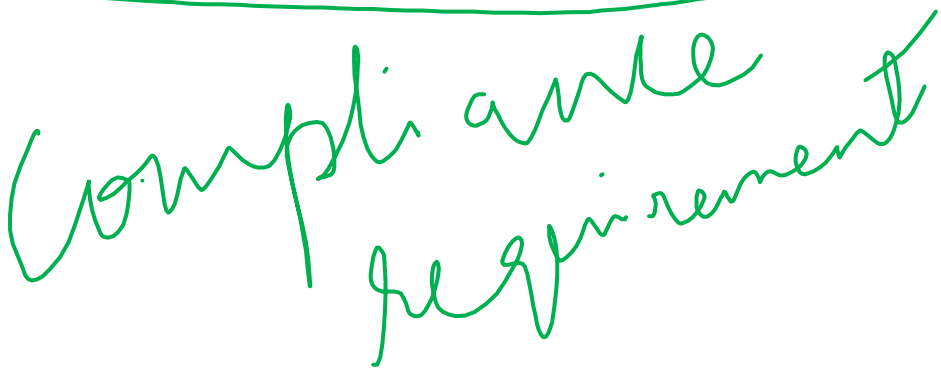
- A system that checks the subject's permissions to an object only once can invite attackers to exploit that system.
- According to Bishop, When a UNIX process tries to read a file, the operating system determines if the process is allowed to read the file. If so, the process receives a file descriptor encoding the allowed access. Whenever the process wants to read the file, it presents the file descriptor to the kernel. The kernel then allows the access. If the owner of the file disallows the process permission to read the file after the file descriptor is issued, the kernel still allows access. This scheme violates the principle of complete mediation, because the second access is not checked. The cached value is used, resulting in the denial of access being ineffective.

Psychological Acceptability

- Accessibility to resources should not be inhibited by security mechanisms. If security mechanisms hinder the usability or accessibility of resources, then users may opt to turn off those mechanisms.
  - Where possible, security mechanisms should be transparent to the users of the system or at most introduce minimal obstruction. Security mechanisms should be user friendly to facilitate their use and understanding in a software application.
- Configuring and executing a program should be as easy and as intuitive as possible, and any output should be clear, direct, and useful.
  - If security-related software is too complicated to configure, system administrators may unintentionally set up the software in a non-secure manner.
- Similarly, security-related user programs must be easy to use and output understandable messages.

Promoting Privacy

- Protecting software systems from attackers that may obtain private information is an important part of software security.
- Many users consider privacy a security concern. Try not to do anything that might compromise the privacy of the user.



BITS Pilani

Pilani | Dubai | Goa | Hyderabad

Secure Architectural Patterns

Secure Patterns

- A software design pattern is a general repeatable solution to a recurring software engineering problem.
- Secure design patterns a general solution to a security problem that can be applied in many different situations.
- Secure design patterns are not restricted to object-oriented design approaches but may also be applied, in many cases, to procedural languages.

Secure Architectural-level Patterns

Architectural-level patterns focus on the high-level allocation of responsibilities between different components of the system and define the interaction between those high-level components.

- Architectural-level Patterns
  - Distrustful Decomposition
  - Privilege Separation (PrivSep)
  - Defer to Kernel

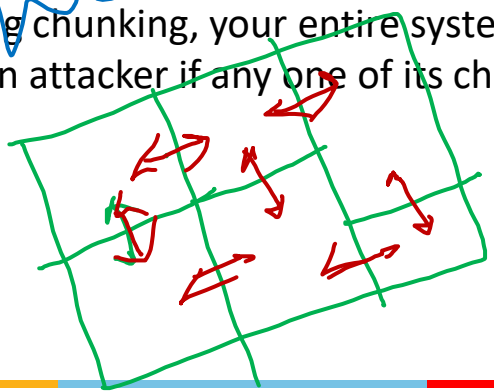
https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=9115

Distrustful Decomposition

Separate the functionality of your software into *mutually untrusting chunks*, so as to shrink the attack windows into each chunk

- Design each chunk under the assumption that other software chunks with which it interacts have been attacked, and it is attacker software rather than normal application software that is running in those interacting chunks.
- Do not expose your data to other chunks via shared memory.

As a result of mutually untrusting chunking, your entire system will not be given into the hands of an attacker if any one of its chunks has been compromised



Distrustful Decomposition (cont..)

Motivation : Many attacks target vulnerable applications running with elevated permissions

Some examples of this class of attack are

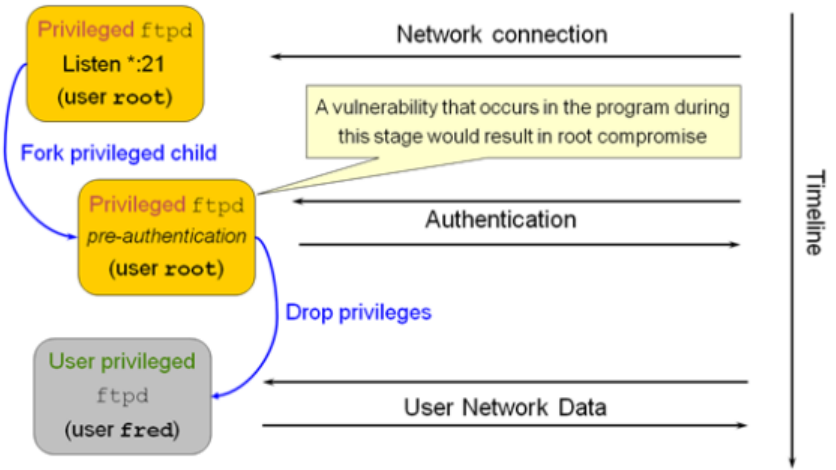
- Attacks in which versions of IE browser running in an account with administrator privileges is compromised
- Security flaws in Norton AntiVirus 2005 that allowed attackers to run arbitrary VBS scripts when running with administrator privileges
- A buffer overflow vulnerability in BSD-derived telnet daemons that allows an attacker to run arbitrary code as root

Consequences : Prevents an attacker from compromising an entire system in the event that a single component program is successfully exploited because no other program trusts the results from the compromised one

Privilege Separation (PrivSep)

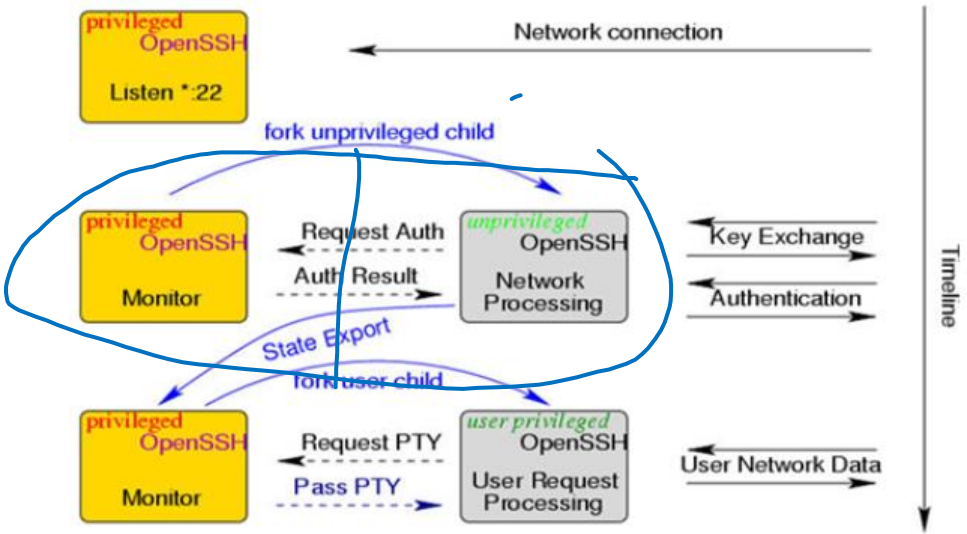
- The PrivSep pattern is a more specific instance of the Distrustful Decomposition pattern.
- Keep to a minimum the part of your code that executes with special privilege.
- If an attacker succeeds in breaking into software that's running at a high level of privilege, the attacker will be operating at a high level of privilege too. That'll give him an extra-wide open "attack window" into your system
- The pattern is applicable if the system performs a set of functions do *not* require elevated privileges, but have relatively large attack surfaces (e.g. communication with untrusted sources, potentially error-prone algorithms)

Privilege Separation (cont..)



Here is a vulnerable implementation where a privileged process is trying to authenticate an unauthenticated user

Privilege Separation (cont..)



- The implementation as per PrivSep pattern.
- The interactions with user and authentication are moved into an unprivileged process.

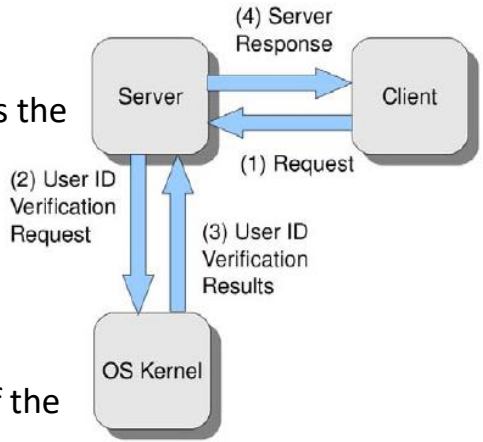
Defer to Kernel

The intent separate “functionality that requires elevated privileges” from “functionality that does not require elevated privileges and to take advantage of existing user verification functionality available at the kernel level.

Designers tend to take control of authorization functionality into their hands. The pattern discourages the tendency

The pattern is applicable to systems:

- That run by users who do not have elevated privileges;
- Where some (possibly all) of the functionality of the system requires elevated privileges; or
- Where the system must verify that the current user is authorized to execute any functionality that requires elevated privileges



BITS Pilani

Pilani | Dubai | Goa | Hyderabad

Secure Design Patterns

Classes of Patterns

Design-level patterns. Design-level patterns describe how to design and implement pieces of a high-level system component, that is, they address problems in the internal design of a single high-level component, not the definition and interaction of high-level components themselves.

- Secure Factory
- Secure Strategy Factory
- Secure Builder Factory
- Secure Chain of Responsibility
- Secure State Machine
- Secure Visitor

https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=9115

Classes of Patterns

Implementation-level patterns. Implementation-level patterns address low-level security issues. Patterns in this class are usually applicable to the implementation of specific functions or methods in the system. Implementation-level patterns address the same problem set addressed by the CERT Secure Coding Standards

- Secure Logger
- Clear Sensitive Information
- Secure Directory
- Input Validation

Secure Factory

- Secure Factory secure design pattern is a security specific extension of the Abstract Factory pattern
- The Secure Factory secure design pattern is applicable if
  - The system constructs different versions of an object based on the security credentials of a user/operating environment.
  - The available security credentials contain all of the information needed to select and construct the correct object.

Secure Strategy Pattern

- The strategy pattern enables an algorithm's behavior to be selected at runtime
- Strategy pattern provides a means to define a family of algorithms, encapsulate each one as an object, and make them interchangeable during runtime
- a class that performs validation on incoming data may use a strategy pattern to select a validation algorithm based on the type of data, the source of the data, user choice, or other discriminating factors
- The secure strategy object performs a task based on the security credentials of a user or environment

Secure Builder Factory

- Secure Builder Factory design pattern is to separate the security dependent rules, involved in creating a complex object, from the basic steps involved in the actual creation of the object.
  - Identify a complex object whose construction depends on the level of trust associated with a user or operating environment. Define the general builder interface using the Builder pattern for building complex objects of this type.
  - Implement the concrete builder classes that implement the various trust level specific construction rules for the complex object.

Secure Chain of Responsibility

- The intent of the Secure Chain of Responsibility pattern is to decouple the logic that determines user/environment-trust dependent functionality from the portion of the application requesting the functionality, make it relatively easy to dynamically change the user/environment-trust dependent functionality.

Motivation

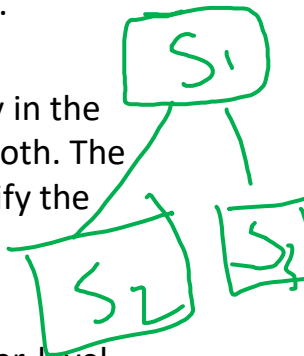
In an application using a role-based access control mechanism, the behavior of various system functions depends on the role of the current user

Consequence

The security-credential dependent selection of the appropriate specific behavior for a general system function logic is hidden from the portions of the system that make use of the general system function.

Secure State Machine

The intent of the Secure State Machine pattern is to allow a clear separation between security mechanisms and user-level functionality by implementing the security and user-level functionality as two separate state machines.



Motivation

Intermixing security functionality and typical user-level functionality in the implementation of a secure system can increase the complexity of both. The increased complexity makes it more difficult to test, review, and verify the security properties of the implementation.

Consequences

- Can test and verify the security mechanisms separately from the user-level functionality
- New security implementation could be implemented with lesser effort

Secure Visitor

Secure systems may need to perform various operations on hierarchically structured data where each node in the data hierarchy may have different access restrictions. The pattern idea is to incorporate security mechanism in data node rather than visitor code.

Motivation

Secure Visitor pattern allocates all of the security considerations to the nodes in the data hierarchy, leaving developers free to write visitors that only concern themselves with user-level functionality

Consequences

The use of this pattern requires that the nodes in the data hierarchy, not the visitors themselves, implement security

Secure Logger

- The intent of the Secure Logger pattern is to prevent an attacker from gathering potentially useful information about the system from system logs and to prevent an attacker from hiding their actions by editing system logs.
- The Secure Logger pattern is applicable if
  - The system logs information to a log file or some other form of logging subsystem.
  - The information contained in the system log could be used by an attacker to devise attacks on the system.
  - System logs are used to detect and diagnose attacks on the system.

Clear Sensitive Information

It is possible that sensitive information stored in a reusable resource may be accessed by an unauthorized user or adversary if the sensitive information is not cleared before freeing the reusable resource. The use of this pattern ensures that sensitive information is cleared from reusable resources before the resource may be reused.

Reusable resources include things such as the following:

- dynamically allocated memory
- statically allocated memory
- automatically allocated (stack) memory
- memory caches
- disk
- disk caches

Secure Directory

The intent of the Secure Directory pattern is to ensure that an attacker cannot manipulate the files used by a program *during* the execution of the program.

–The Secure Directory pattern ensures that the directories in which the files used by the program are stored can only be written (and possibly read) by the user of the program.

The Secure Directory pattern is applicable for use in a program if

- The program will be run in an insecure environment; that is, an environment where malicious users could gain access to the file system used by the program.
- The program reads and/or writes files.
- Program execution could be negatively affected if the files read or written by the program were modified by an outside user while the program was running.

The program should check that a directory offered to it is secure, and refuse to use it otherwise. Implementation of the Secure Directory pattern involves the following steps:

- Find the canonical pathname of the directory of the file to be read or written.
- Check to see if the directory, as referenced by the canonical pathname, is secure.
  - If the directory is secure, read or write the file.
  - If the directory is not secure, issue an error and do not read or write the file.

Input Validation

Input validation requires that a developer correctly identify and validate all external inputs from untrusted data sources

Motivation

- The use of unvalidated user input is the root cause of many serious security exploits, such as buffer overflow attacks, SQL injection attacks, and cross-site scripting attacks.
- In a client-server architecture, it is problematic if only client-side validation is performed. It is easy to spoof a web page submission and bypass any scripting on the original page

False positives

Software Composition Analysis

vs

False negatives

Software Composition Analysis

- Generally term is used for managing open source component use
- Involves scans of an application's code base to identify all open source components for
  - license compliance data, and
  - security vulnerabilities
- The scan covers identifying
  - direct dependencies &
  - transitive dependencies

Software Composition Analysis

- SCA tools are expected to perform the following with software vulnerabilities
  - Detection
  - Prioritization – identify risk/criticality of vulnerabilities
  - Remediation
- SCA tool requires
  - Comprehensive vulnerability database
  - Ability to patch

Log4j & Open Source Software

innovate achieve lead

Can any SCA tool determine use of vulnerable component?

How often open source software is embedded in enterprise software?

What is the risk of using open source code?

https://www.securitymagazine.com/articles/92368-synopsys-study-shows-91-of-commercial-applications-contain-outdated-or-abandoned-open-source-components

https://scantist.com/resources/blog/log4j-software-composition-analysis

https://www.forrester.com/blogs/log4j-open-source-maintenance-and-why-sboms-are-critical-now/

Network Security

Modern Network Security Architecture

Large number of options available for organizational networks today means new requirements for network security

- Public Cloud
- Private Cloud
- Hybrid Cloud
- On-Premises

Each of the above come with wide variety of options and tools

Modern Network Security Architecture

- Zero-Trust Architecture (ZTA) is a network security paradigm that operates from the assumption that some actors on the network are hostile, and there are too many entry points to fully protect.
- Network firewall is aimed at preventing anyone from directly accessing the network servers that host an organization's applications and data
- Microsegmentation inhibits an attacker already on the network from moving laterally within it to access critical assets
- A secure web gateway (SWG) is responsible for connecting the user to the desired website and perform functions such as URL filtering, malicious content inspection, web access controls etc.
- DNSSEC strengthens authentication in DNS using digital signatures based on public key cryptography.

https://www.gartner.com/en/articles/17-network-security-concepts-and-acronyms-you-need-to-know

Modern Network Security Architecture

- Secure Access Service Edge (SASE) is an emerging framework that combines comprehensive network security functions, such as SWG, SD-WAN and ZTNA

- VDI and DaaS are both solutions for hosted desktops, the key difference between the two is that VDI is managed by the company itself, hosted and managed on-site or at a colocation facility, while DaaS is a managed infrastructure service, delivered by a provider and hosted in their data centres.
- Network Security Policy Management (NSPM) involves analytics and auditing to optimize the rules that guide network security, as well as change management workflow, rule-testing and compliance assessment and visualization. NSPM tools may use a visual network map that shows all the devices and firewall access rules overlaid onto multiple network paths.

Software Security Engineering, Julia H. Allen, et al, Pearson, 2008.
Security in Computing by Charles P. Pfleeger, Shari L. Pfleeger, and Deven Shah Pearson Education 2009
Computer Security: Principles and Practice by William Stallings, and Lawrie Brown Pearson, 2008.
www.owasp.com
www.microsoft.com
https://resources.sei.cmu.edu/asset_files/TechnicalReport/2009_005_001_15110.pdf

Thank You!