PHP Security

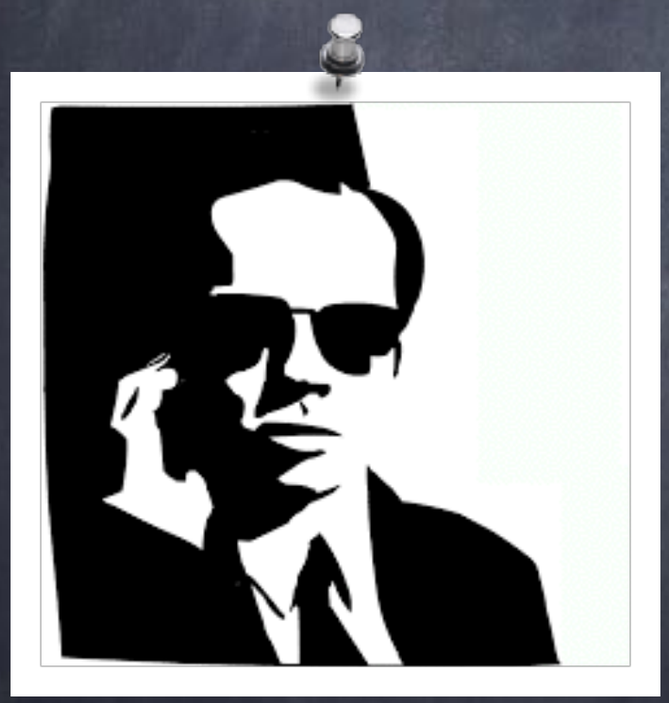
Jupiter Zhuo

"A **vulnerability** in an application will allow a malicious user to exploit a **network** or a **host**."

-- Carlos Lyons, Corporate Security, Microsoft

What's The Threat

Threat "Agent" Or "Actor"



Internal

Partner

External

Types Of Threats

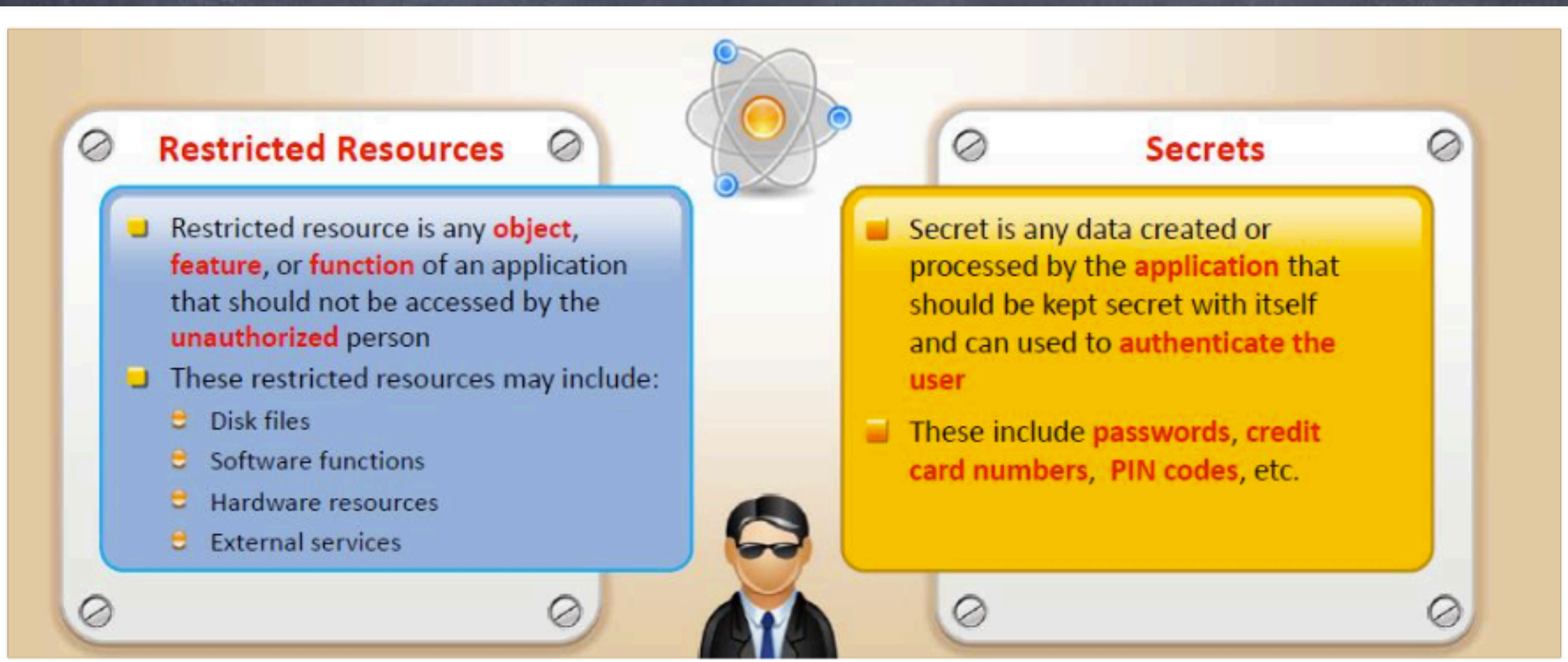
Software

People

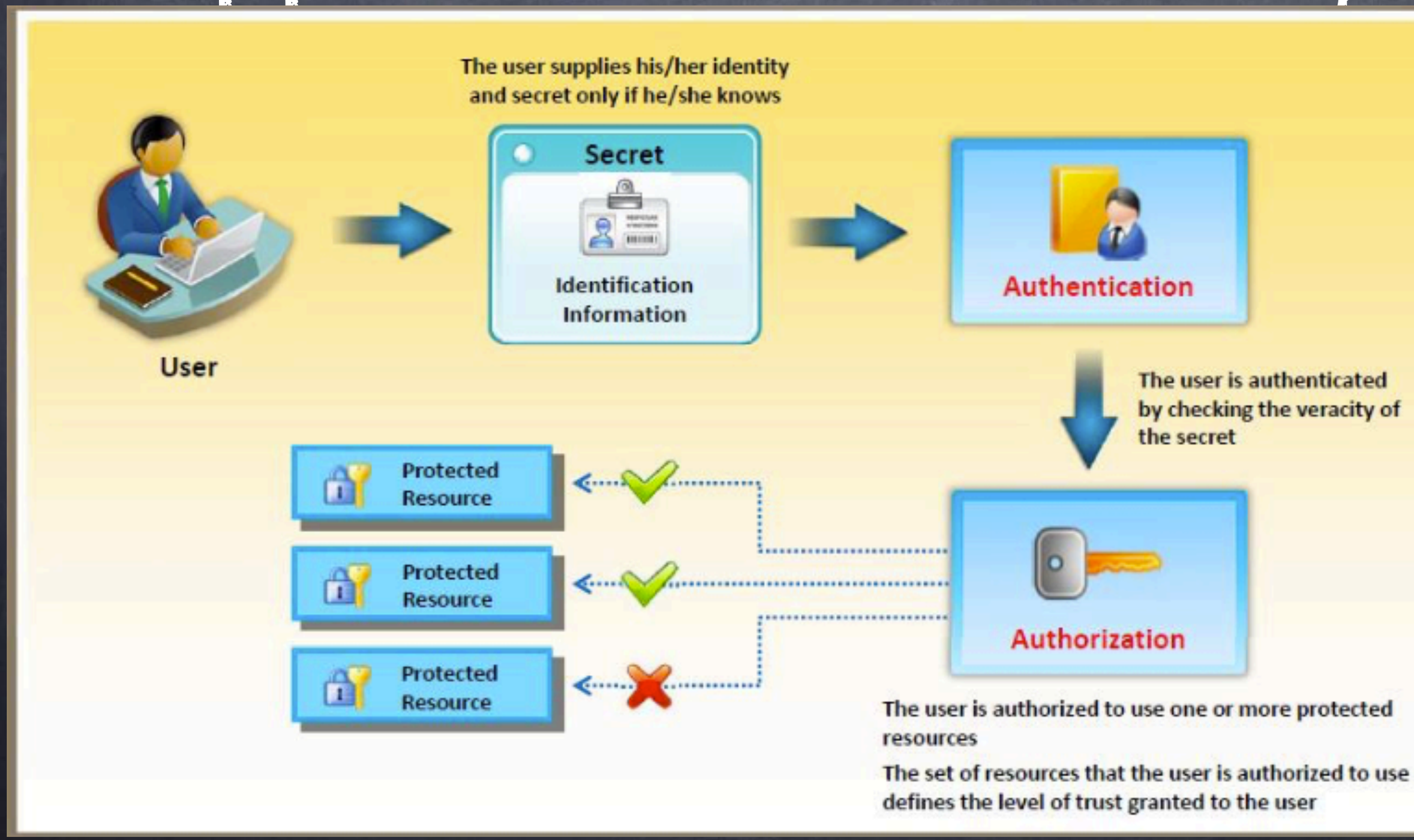
Criminal

Understanding Application Security

Any Application is said to be secured when it's restricted resources and secrets are protected from malicious use



Understanding Application Security



What's Secure Coding

Secure coding is a state of practice for developing **robustly secure software** in such a way that it should withstand any type of possible attacks

Secure coding practices help in protecting data from **theft or corruption** due to logical exploitation and program crashes

An insecure program can be exploited by an attacker to take control of a **server or a user's computer** and launch different types of host/network attacks such as denial-of-service, identity theft, malware attacks, etc.

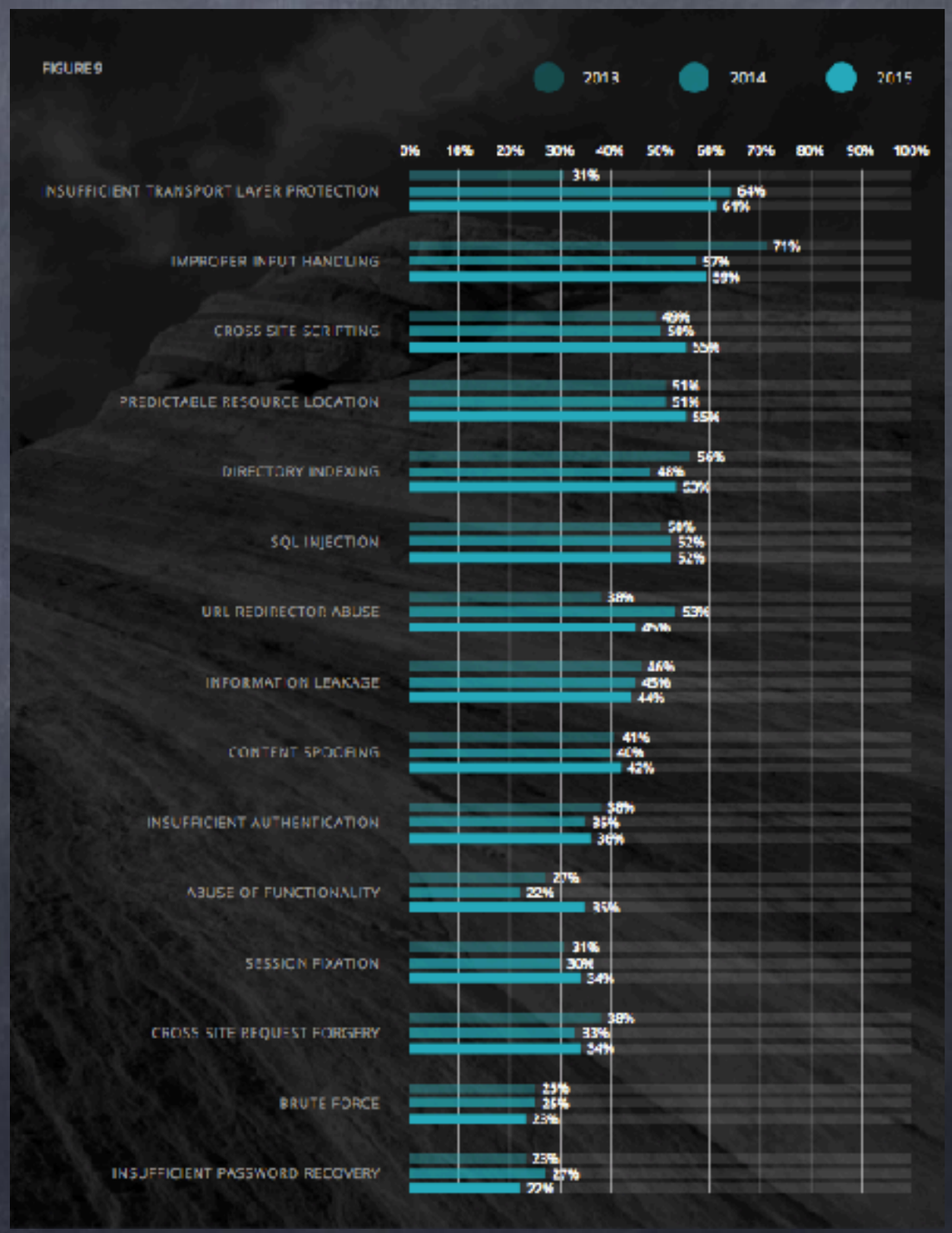
According to the CERT/CC vulnerability reports, small programming errors can cause serious **vulnerabilities** in the application

Why are security Mistakes made ?

- Improper Application of security principles (Privilege) at the architecture stage of the entire code
- Most of programming curriculum often don't include Security issues
- Improper handling of flawed input data at design stage
- Missing some of the flaws at the testing stage, because improper testing
- Minor Flaws in the code given space to various vulnerabilities, resulting in serious damage to system
- Writing Secure code required extra cost, time and effort
- no proper guidance on secure coding to code developers at every stage of the project development

Web Application Attack Scenario

Source : https://info.whitehatsec.com/rs/675-YBI-674/images/WH-2016-Stats-Report-FINAL.pdf



OWASP TOP 10 Web Application Critical Security Risk

OWASP Top 10 – 2013 (Previous)	OWASP Top 10 – 2017 (New)
A1 – Injection	A1 – Injection
A2 – Broken Authentication and Session Management	A2 – Broken Authentication and Session Management
A3 – Cross-Site Scripting (XSS)	A3 – Cross-Site Scripting (XSS)
A4 – Insecure Direct Object References - Merged with A7	A4 – Broken Access Control (Original category in 2003/2004)
A5 – Security Misconfiguration	A5 – Security Misconfiguration
A6 – Sensitive Data Exposure	A6 – Sensitive Data Exposure
A7 – Missing Function Level Access Control - Merged with A4	A7 – Insufficient Attack Protection (NEW)
A8 – Cross-Site Request Forgery (CSRF)	A8 – Cross-Site Request Forgery (CSRF)
A9 – Using Components with Known Vulnerabilities	A9 – Using Components with Known Vulnerabilities
A10 – Unvalidated Redirects and Forwards - Dropped	A10 – Underprotected APIs (NEW)

Common Security Threats

Category	Vulnerability	Attacks
Input and Data Validation	• Non-validated input usage in the HTML output stream • Non-validated input usage generating SQL queries • Dependence on client-side validation • Usage of input files names, URLs, or usernames for security decisions • Application-only filters usage for malicious input • Trusting data read from databases, file shares, network resources, etc. • Non-validation of input from all sources such as cookies, query string parameters, HTTP headers, etc.	• Buffer overflow • Cross-site scripting • SQL injection • Cross-site request forgery • Canonicalization attacks • Query string manipulation • Form field manipulation • Cookie manipulation • HTTP header manipulation
Authentication	• Using weak passwords • Mixing personalization with authentication • Insecure authentication form settings • Allowing over-privileged accounts • Storing clear text information in configuration files • Passing clear text information over the network • Allowing prolonged session lifetime	• Credential theft • Eavesdropping • Brute-force and dictionary attacks • Cookie replay attacks • Account hijacking • Information leakage
Privacy and Data Compromise	• Passing vital data in clear text over networks • Storing secrets unnecessarily • Storing secrets in code and clear text • Access to sensitive data in storage	• Accessing sensitive data in storage and memory • Data tampering • Network eavesdropping • Information disclosure

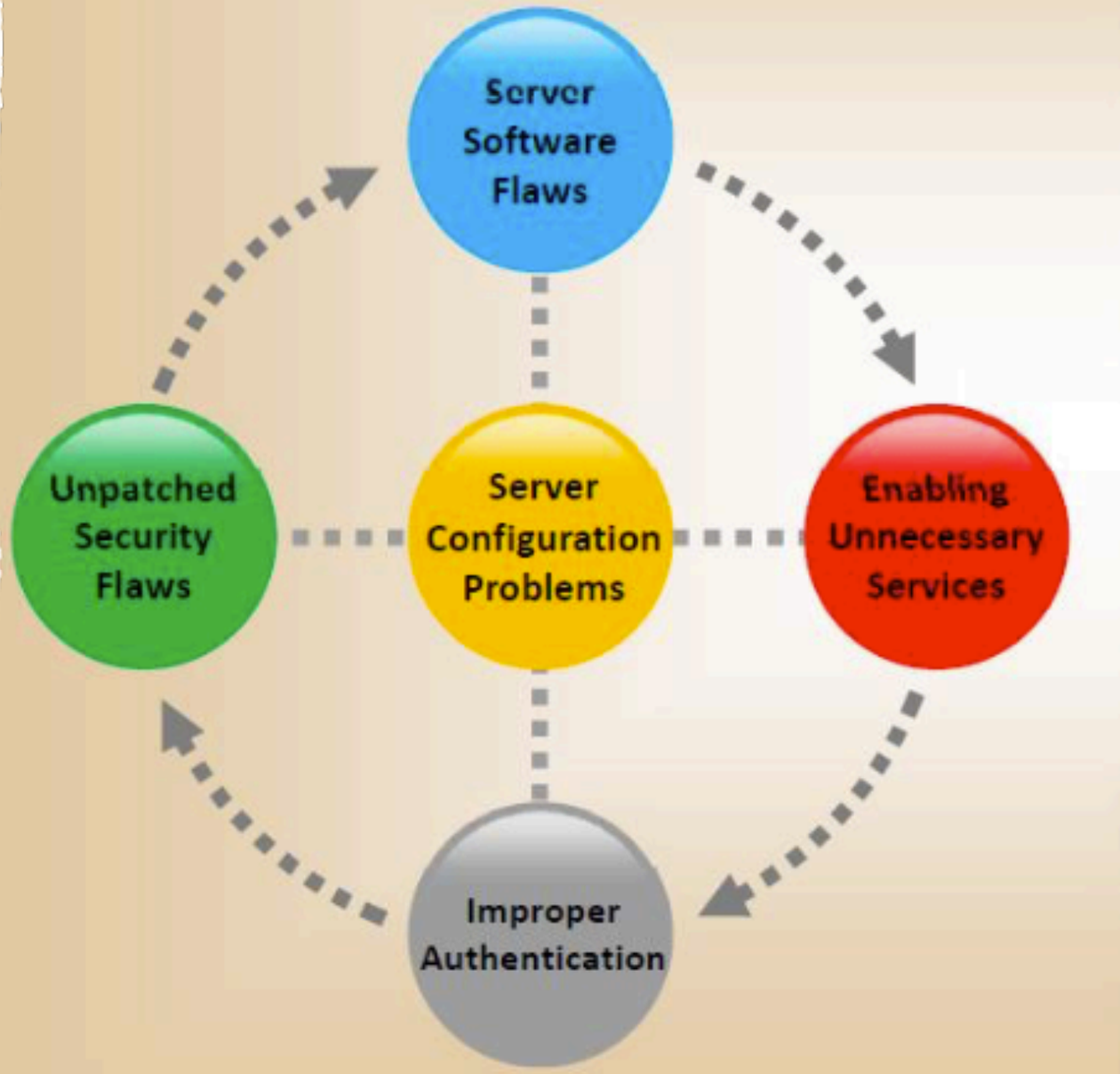
Common Security Threats

Category	Vulnerability	Attacks
Authorization	• Dependence on a single gatekeeper • Insufficient separation usage of privileges • Not providing locks to system resources for particular application • Failed to restrict database access for particular stored procedures	• Privilege escalation • Disclosure of confidential data • Data tampering • Luring attacks
Session Management	• Inserting session identifiers in query strings • Using unencrypted channels for passing session identifiers • Allowing extended session lifetime • Insecure session state stores	• Session hijacking • Session fixation • Session replay • Man-in-middle attacks
Exception Management	• Not succeeding in using structured exception handling • Providing detailed error information to the user	• Disclosure of sensitive information • Denial-of-service attacks
Application Configuration	• Using process and service accounts with higher privileges	• Salvage of clear text configuration data

Common Security Threats

Category	Vulnerability	Attacks
Configuration Management	• Insecure administration interface usage • Over-privileged process and service accounts utilization • Insecure configuration stores usage • Stores clear text configuration information • Too many administrators	• Illegal access to administration Interfaces • Illegal access to configuration stores • Salvage of clear text configuration secrets
Cryptography	• Weak encryption • Insecure distribution of keys • Not succeeding in secure encryption keys • Lack of robust key generation or key management • Custom cryptography usage • Wrong algorithm usage or utilizing a key size that is too small • Utilization of the same key for an extended period of time	• Loss of decryption keys • WEP Encryption cracking
Auditing and Logging	• Not succeeding in auditing failed logons • Not succeeding in securing audit files • Not succeeding auditing across application tiers	• Denial of performing an operation by user • Exploitation of an application by an attacker without trace • Covering tracks
Machine Threats	• Using weak passwords • Allowing prolonged session lifetime • Not using anti-virus and anti-spyware	• Virus, Trojan, worm attacks • Arbitrary code execution • Password attacks • Unauthorized access to machine

Security Misconfiguration



- **Easy Exploitation**

  Using misconfiguration vulnerabilities, attackers **gain unauthorized accesses** to default accounts, read unused pages, exploit unpatched flaws, and read or write unprotected files and directories, etc.

- **Common Prevalence**

  The mistakes in configuring security can take place at any of the application stack levels such as platform, **web server**, application server, framework, and **custom code**

- **Example**
  - The vulnerability exists at the application server level when its **Admin console** is not removed as it automatically get installed
  - Failed to change **default accounts** such as default passwords and usernames
  - Attacker may use standard Admin console on server and can commit the attacks with **default passwords**

Cross Site Scripting XSS

- Cross-site scripting ('XSS' or 'CSS') attacks exploit **vulnerabilities in dynamically generated web pages** in which an attacker injects the malicious script into input fields or in the URL that runs once users visit that particular web page
- It occurs when invalidated input data is included in **dynamic content** that is sent to the user's **web browser** for rendering
- Attackers inject malicious **JavaScript**, VBScript, ActiveX, HTML, or **Flash** for execution on a victim's system by hiding it within **legitimate requests**

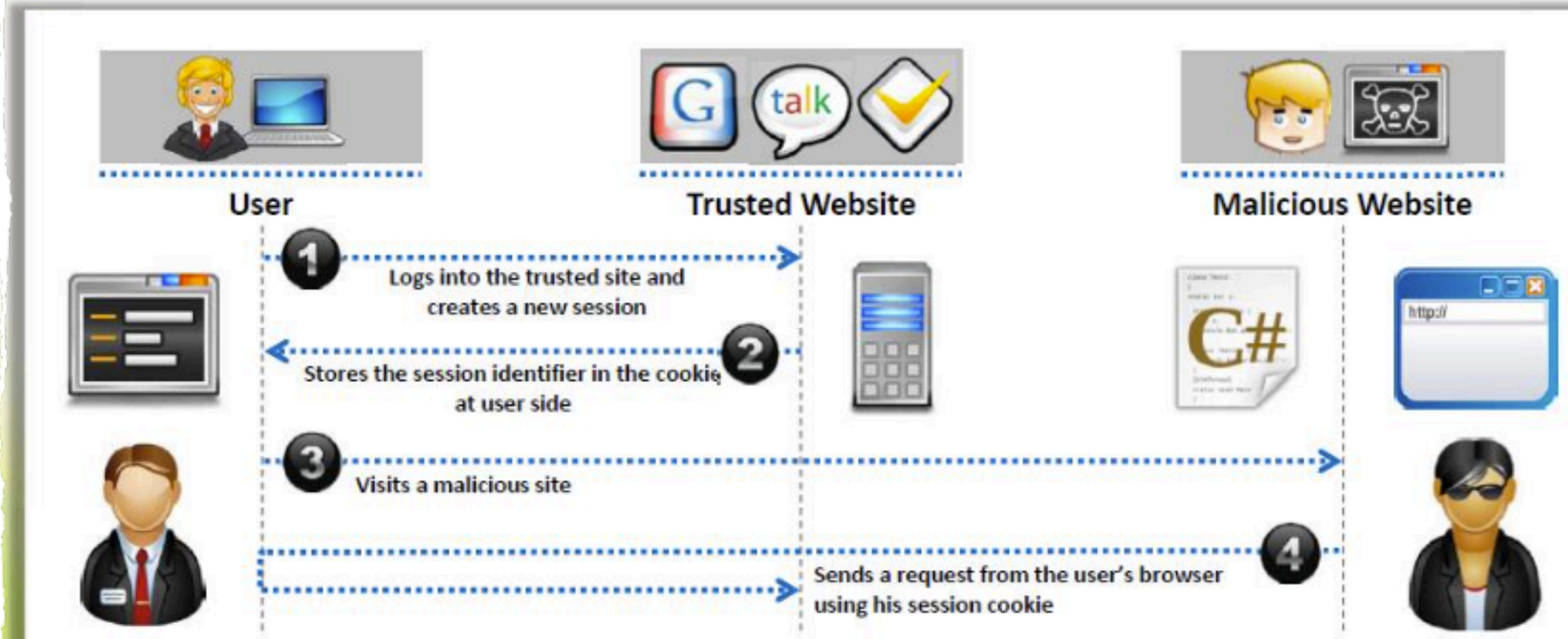
- Malicious script execution
- Redirecting to a **malicious server**
- Exploiting user privileges
- Ads in **hidden IFRAMES** and pop-ups
- Data manipulation and session hijacking
- Brute force password cracking
- Data theft and **intranet probing**
- Keylogging and remote monitoring

SQL Injection Attacks

- SQL injection attacks use a **series of malicious SQL queries** to directly manipulate the database
- An attacker can use a vulnerable web application to **bypass normal security measures** and obtain direct access to the valuable data
- SQL injection attacks can often be executed from the **address bar**, from within **application fields**, and through **queries and searches**

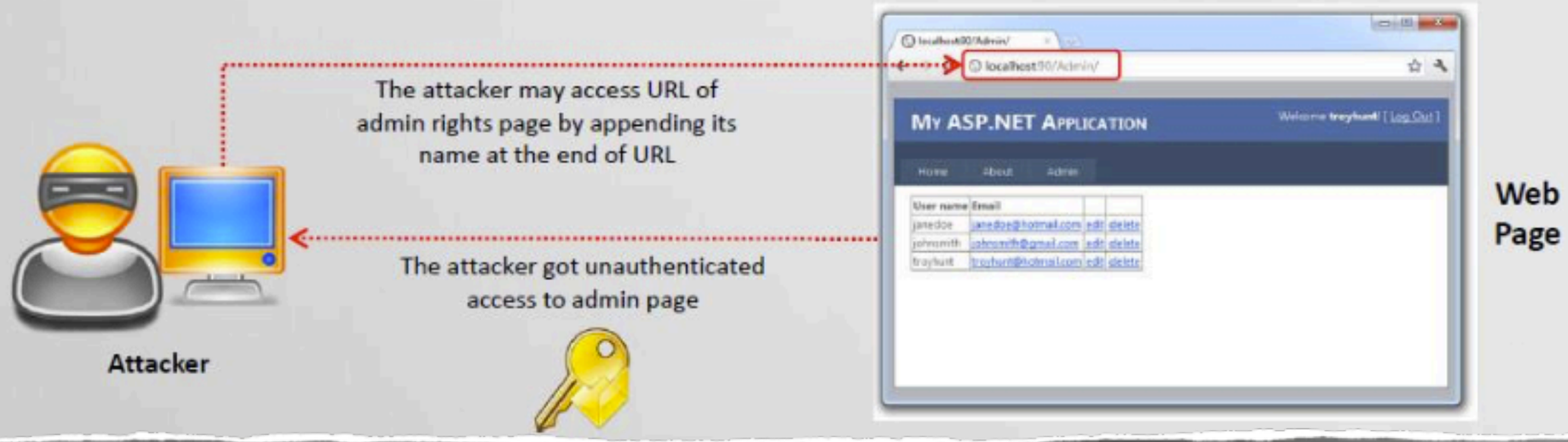
Cross Site Request Forgery (CSRF Attack)

- Cross-Site Request Forgery (CSRF) attacks exploit web page vulnerabilities that allow **an attacker to force an unsuspecting** user's browser to send malicious requests they did not intend
- If user establishes a session with the trusted site and if he simultaneously visits a **malicious site**; then a malicious HTTP request is sent to the trusted site using user session that was established with the trusted site, this results in compromising the **integrity** of the application



Failure to restrict URL Access

- There are some pages in the web application that need only authentication access such as **administrator rights**
- The **developer** tries to hide these pages from others by not providing direct links to these pages
- The developer create these pages as **hidden pages** that do not have a link pointing to them
- The attacker uses **forceful browsing techniques** to find the URL of these sensitive pages to gain **unauthenticated access**
- The client-side access privilege checking is also **vulnerable** to this type of attack as the attacker can easily bypass **client side** check



Insufficient Transport Layer Protocol

Supports Weak Algorithm

- Insufficient transport layer protection supports weak algorithms, uses **expired** or **invalid** certificates

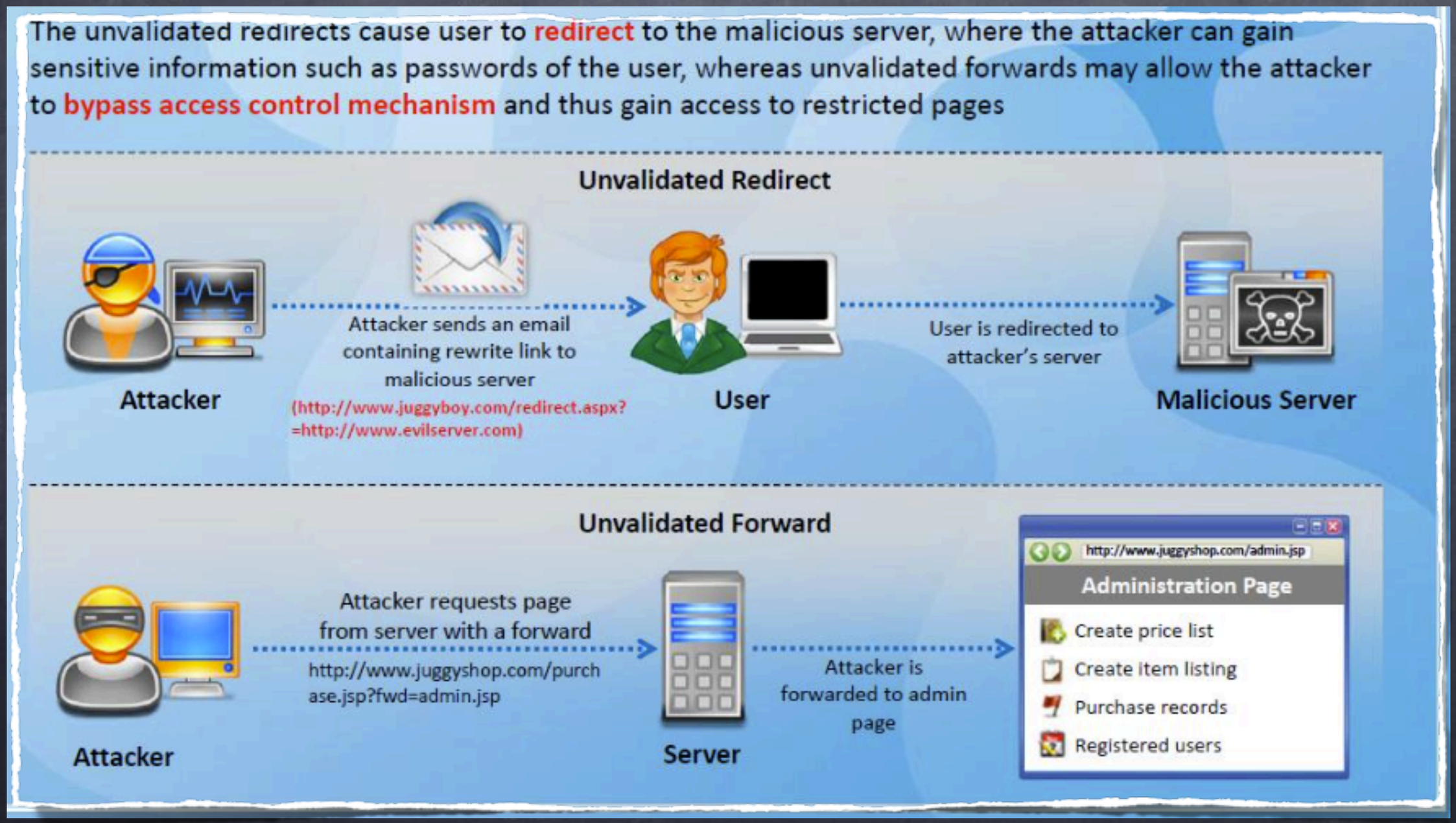
Exposes Data

- This vulnerability exposes user's data to **untrusted third-parties** and can lead to account theft

Launch Attacks

- Underprivileged SSL setup can also help attacker to launch **phishing** and **MITM** attacks

Unvalidated Redirects and Forwards



Insecure Direct Object References

Direct object reference attacks occur when a developer uses **internal object implementation** directly on the user interface

These objects may include **file**, **directory**, **database record**, or **key**

Direct Object References

Attacker gets the idea about internal implementation and may use these objects as part of **URL** to access and modify sensitive information without any authorization

In bank application, attacker may use **account number** directly as an input field in the user interface, as he/she knows that account number is generally the primary key of any bank application

This may give path to the attacker to commit attacks on the **bank database**

Attacker

Web Interface

Database

Broken Authentication and session management

An attacker uses vulnerabilities in the authentication or session management functions such as exposed **accounts**, **session IDs**, **logouts**, password management, **timeouts**, remember me, **secret questions**, account updates, and others to impersonate users

Session ID in URLs

Attacker **sniffs the network traffic** or tricks the user to get the session IDs, and reuses the session IDs for malicious purposes

http://juggyshop.com/sale/saleitems=304;jsessionid=12OMTOIDPXM0OQSABGCKLHCJUN2JV?dest=New Mexico

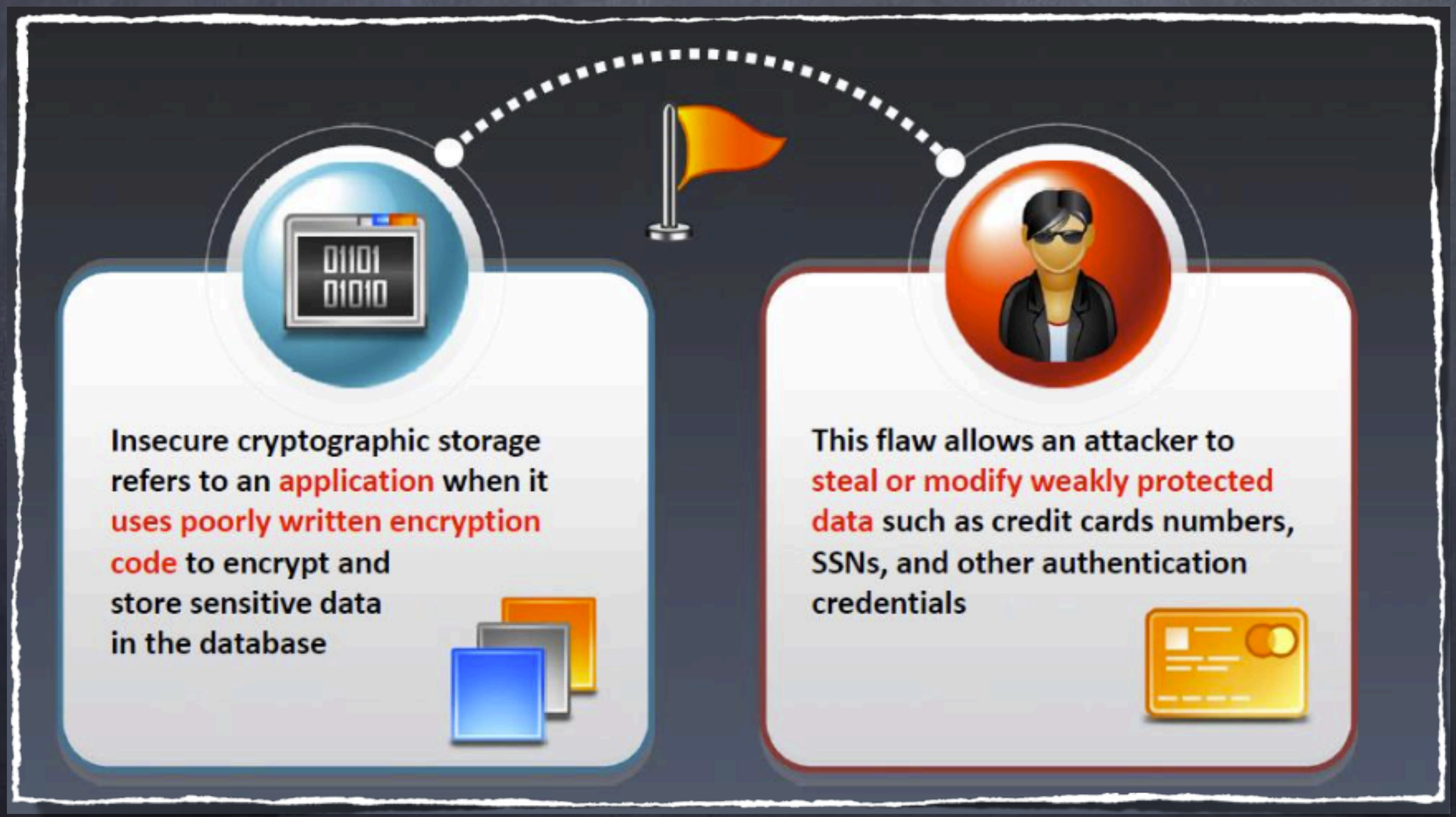
Timeout Exploitation

If an application's timeouts are not set properly and a user simply **closes the browser without logging out** from sites accessed through a public computer, attacker can use the same browser later and exploit the user's privileges

Password Exploitation

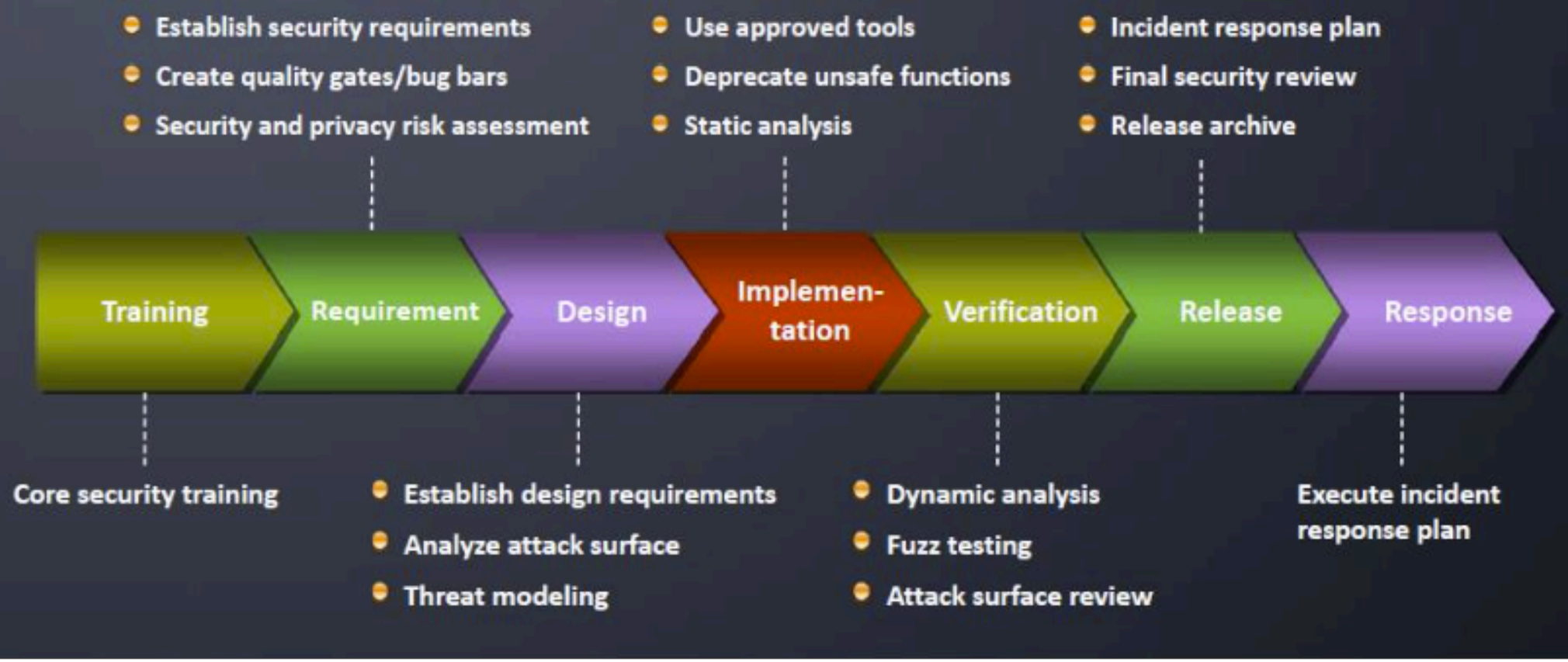
Attacker gains access to the web application's password database. If user **passwords are not encrypted**, the attacker can exploit every user's password

Insecure Cryptographic Storage



Secure Development Lifecycle (SDL)

- Security Development Lifecycle is a program developed by the Microsoft for **developing secure applications**
- This program is divided into **seven phases** and provides information about **security practices**, **guidelines**, and **technologies**



Phases of SDL

1 Training

- The appropriate security implementation training is given to the people involved in the **software development process**
- These people may include the **developer**, **tester**, **program manager**, etc.

2 Requirements

- The security requirements are specified for a software project during this phase, which allows the developer to **identify the key security milestones** in the project
- The **quality gates** and **bug bars** help in providing security and privacy quality up to that level where it can be acceptable
- Security and privacy risk assessment is also performed in this phase

3 Design

- Most of the **security and privacy concerns** can be eliminated in the design phase of software development
- The design requirements are specified to **reduce attack surface** on the software
- **Threat modeling** is performed in this phase

4 Implementation

- These **tools** are used to check the security of the project developed
- These tools are defined and published by the development team and get approved from the **security advisor**
- The APIs and functions that can be vulnerable to the attack are **deprecated**
- Static analysis is performed to **analyze the source code** prior to compilation

Phases of SDL (Cont'd)

- Dynamic program analysis is done by using **fuzz testing technique** to ensure that a program is working as designed
- Tools are used to **monitor application behavior** for security problems

5

Verification

- The incident response plan is created for **known and unknown vulnerabilities** at the time of release
- A **Final Security Review (FSR)** is performed before releasing application

6

Release

- The previously defined incident response plan is executed if any **vulnerability** occurs, which may create threat to the application

7

Response

Integrating Security into the Development Lifecycle

- The process of integrating security into the application development lifecycle, negotiating the **policies**, **risks**, and the **requirements of the application** in such a way that the application should be secured and its performance should not be degraded
- Eight steps for integrating security into application development lifecycle are:

1

Initial Review

- During this phase, the security team understand the **purpose**, **technical environment**, **processes and procedures**, etc. of the application and asses the initial risks on it
- This initial review is done at the **requirement phase** of software development lifecycle

2

Threat Modeling

- The security team performs the threat modeling to **identify the sensitive area of the application**
- It helps them to identify the area of application where **added security attention is required**
- It is **performed at early stages** of the software development life cycle

3

Design

- The application design reviews are performed in order to **identify the potential risk**
- This is done at design phase of the SDLC to **minimize attack surface**

4

Development

- A security testing is performed after finishing **unit testing** by the developer
- This includes **reviewing code** for best security practices

Integrating Security into the Development Lifecycle (Cont'd)

5

Deployment: Risk assessment

After performing all the security reviews in the respective phases of the software development lifecycle, a risk assessment is done prior to the deployment phase

6

Risk Mitigation

The security controls are prioritized, evaluated, and implemented to **mitigate the vulnerabilities** identified during risk assessment stage

7

Benchmark

The resulting secured application is compared with the **industry standards** to deliver the security scorecard. This allows the developer to determine the **security integration** in software development life cycle whether it is up to the mark or need to improve

8

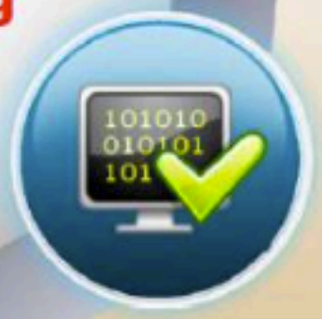
Maintenance

A **periodic security checking** is performed on all critical areas of the application where security controls are provided

Security in the Design Stage: Threat Modeling

Threat modeling is a process of **identifying**, **analyzing**, and **mitigating the threats** to the application

It is performed at the **design phase** of the secure development lifecycle



Threat Modeling

It is a structured approach that allows the developer to **rate the threats** based on architecture and implementation of the application

It is an **iterative process** that starts from the design phase of the application and iterates throughout the application lifecycle until all possible **threats to the applications** are identified

Threat Modeling Process (1 of 13)

1. Identifying valuable assets
2. Creating architecture overview of the application
3. Decomposing the application
4. Identifying threats in the application
5. Documenting the threats
6. Rating the threats

Threat Modeling Process (2 of 13)

Step 1

Identifying Valuable Assets

- This steps involves identifying those assets that require **protection from attack**
- These assets may include resources such as confidential data in the database, file system, or other important resources

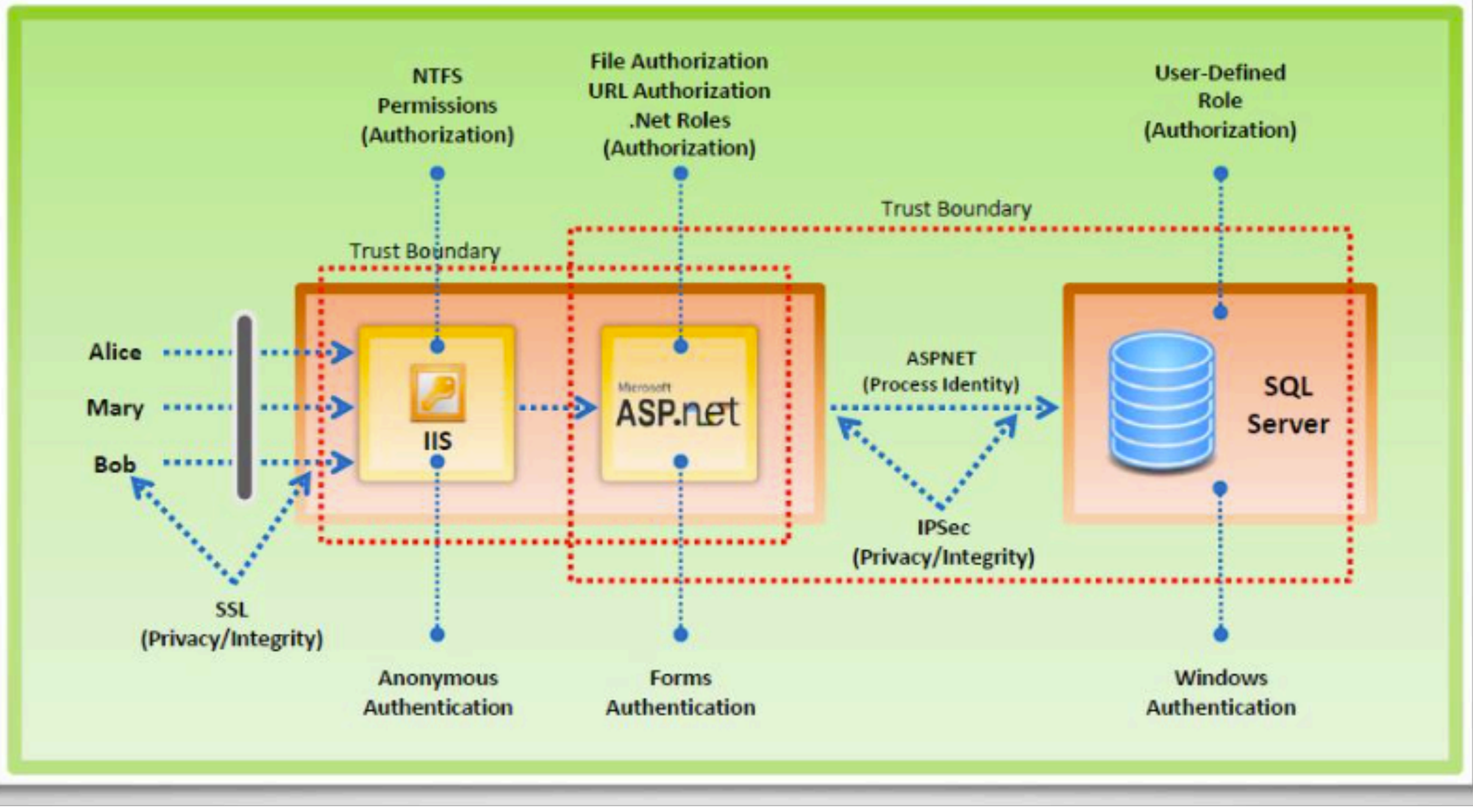
Step 2

Creating Architecture Overview of the Application

- Simple diagrams are created and documented that describe the **working of the application**
- It includes external entities, processes, trust boundaries, data stores, and data flow between different types of functions in the application
- The following tasks are performed in this step:
  - Identifying the function of an application and how it uses or access the assets of the application
  - Creating an architecture diagram for the application
  - Identifying the technologies required to implement solution

Threat Modeling Process (3 of 13)

Simple Application Architecture Diagram



Threat Modeling Process (4 of 13)

Step 3

Decomposing the Application

- The application is decomposed into such a level where it is possible to **identify trust boundaries**, data flow, entry points, and privileged code in the application and recognize the **areas of vulnerabilities** among them
- A **security profile** is created and documented for application depending on the area of vulnerabilities

Application Decomposition

Security Profile

- Input Validation
- Authentication
- Authorization
- Configuration Management
- Sensitive Data
- Session Management
- Cryptography
- Parameter Manipulation
- Exception Management
- Auditing and Logging

- Trust Boundaries
- Data Flow
- Entry Points
- Privileged Code

Threat Modeling Process (5 of 13)

Creating Security Profile:

Category	Considerations
Input Validation	• Is all input data validated? • Could an attacker inject commands or malicious data into the application? • Is data validated as it is passed between separate trust boundaries (by the recipient entry point)? • Can data in the database be trusted?
Authentication	• Are credentials secured if they are passed over the network? Are strong account policies used? • Are strong passwords enforced? • Are you using certificates? • Are password verifiers (using one-way hashes) used for user passwords?
Authorization	• What gatekeepers are used at the entry points of the application? How is authorization enforced at the database? • Is defense in depth strategy used? • Do you fail securely and only allow access upon successful confirmation of credentials?
Configuration Management	• What administration interfaces does the application support? How are they secured? • How is remote administration secured? • What configuration stores are used and how are they secured?
Sensitive Data	• What sensitive data is handled by the application? How is it secured over the network and in persistent stores? • What type of encryption is used and how are encryption keys secured?

Threat Modeling Process (6 of 13)

Creating Security Profile:

Category	Considerations
Session Management	• How are session cookies generated? How are they secured to prevent session hijacking? • How is persistent session state secured? • How is session state secured as it crosses the network? • How does the application authenticate with the session store? • Are credentials passed over the wire and are they maintained by the application? If so, how are they secured?
Cryptography	• What algorithms and cryptographic techniques are used? How long are encryption keys and how are they secured? • Does the application put its own encryption into action? • How often are keys recycled?
Parameter Manipulation	• Does the application detect tampered parameters? Does it validate all parameters in form fields, ViewState, cookie data, and HTTP headers?
Exception Management	• How does the application handle error conditions? Are exceptions ever allowed to propagate back to the client? • Are generic error messages that do not contain exploitable information used?
Auditing and Logging	• Does your application audit activity across all tiers on all servers? How are log files secured?

Threat Modeling Process (7 of 13)

Step 4 Identifying Threats in the Application

Threat Identification

The possible threats in the application are identified in this step, which include **network threats**, **host threats**, and **application threats**

Threat identification process uses two models to examine each aspect of the **security profile** of the application

- **The STRIDE model**
- **The DREAD model**

Threat Identification Models

Threat Modeling Process (8 of 13)

The STRIDE Model

- The STRIDE model categorizes the application threats based on the **goals** and **purpose of the attack** which helps developer to **develop a security strategy**
- It also includes the **countermeasures** for all threat categories
- This model describes the following threat categories:

Desired Property	Threats	Description
Authentication	Spoofing	Unauthorized access to a system by using a false identity
Integrity	Tampering	Code and data modification without authorization
Non-repudiation	Repudiation	Ability of users to claim that they do not perform some actions against the application
Confidentiality	Information disclosure	Unwanted exposure of private data to unauthorized users
Availability	Denial of service	Ability to deny a system or application unavailable to the legitimate users
Authorization	Privilege escalation	Ability of user with limited privileges to elevate their privileges with an application without authorization

Threat Modeling Process (9 of 13)

STRIDE Threats and Countermeasures:

Desired Property	Threats	Countermeasures
Authentication	Spoofing	• Use strong authentication mechanisms • Avoid storing credentials in clear text • Do not pass sensitive data in clear text over networks
Integrity	Tampering	• Use strong authentication mechanisms • Secure authentication tokens with encrypted communication channels • Use digital signature
Non-repudiation	Repudiation	• Identify malicious behavior • Create secure audit trails • Use digital signature
Confidentiality	Information disclosure	• Use strong authorization and encryption mechanism • Avoid storing credentials in clear text
Availability	Denial of service	• Use bandwidth controlling techniques
Authorization	Privilege escalation	• Use limited privileged service accounts

Threat Modeling Process (10 of 13)

Step 5 **Documenting Threats**

A standard template that describes several **threat attributes** including threat description and threat target are used to document the threat

The **risk rate** is left blank as it is to be filled in during the final risk rating step

Threat Description	Attacker obtains authentication credentials by monitoring the network
Threat Target	Web application user authentication process
Risk	
Attack Techniques	Use of network monitoring software
Countermeasures	Use SSL to provide encrypted channel

A sample template for documenting the threats

Threat Modeling Process (11 of 13)

Step 6 **Rating the Threats**

1. This step rates the threats based on the **risks they possess**

2. The high-risk-rated threats are resolves first and so on, depending upon **risk ratings**

3. The formula for calculating risk is:

   RISK = PROBABILITY * DAMAGE POTENTIAL

Threat Modeling Process (12 of 13)

DREAD Model

- DREAD stands for damage potential, reproducibility, exploitability, affected users, and discoverability
- DREAD model is used to rate the various security threats on the application by **calculating risks of each threats**
- The severity levels are assigned to various threats on the application so as to **mitigate these threats as per their severity**

	Rating	High (3)	Medium (2)	Low (1)
D	Damage Potential	The attacker can subvert the security system; get full trust authorization run as administrator; upload content	Leaking sensitive information	Leaking trivial information
R	Reproducibility	The attack can be produced every time and does not require a timing window	The attack can be reproduced, but only with a timing window and a particular race situation	The attack is very difficult to reproduce even with knowledge of the security hole
E	Exploitability	A novice programmer could make the attack in a short time	A skilled programmer could make the attack then repeat the steps	The attacker requires an extremely skilled person and in-depth knowledge every time to exploit
A	Affected Users	All users, default configuration key customers	Some users, non-default configuration	Very small percentage of users, obscure feature affects anonymous users
D	Discoverability	Published information explains the attack. The vulnerability is found in the most commonly used feature and is very noticeable	The vulnerability in a seldom-used part of the product and only a few users should come across it. It would take some thinking to see malicious use	The bug is obscure and it is unlikely that users will work out damage potential

Threat Modeling Process (13 of 13)

The DREAD Model

Threat	D	R	E	A	D	Total	Rating
Attacker obtains authentication credentials by monitoring the network	3	3	2	2	2	12	High
SQL commands injected into application	3	3	3	3	2	14	High

Sample DREAD Rating Table

Threat Description	Attacker obtains authentication credentials by monitoring the network
Threat target	Web application user authentication process
Risk rating	High
Attack techniques	Use of network monitoring software
Countermeasures	Use SSL to provide encrypted channel

A sample template for documenting the threats after Risk Rating

Guidelines for Applying Security in Implementation Phase of SDL

- Do not use the **unsafe** or **deprecated function**
- Use the updated **compiler tools**
- Use **static** and **dynamic** analysis tools
- Perform the manual **code review**
- Validate all the **inputs** coming from the user
- Encode **outputs**
- Use anti-cross site scripting **libraries**
- Do no use **string concatenation** for dynamic SQL
- Eliminate weak **encryption**

Five good habits of a security

- Nothing is 100% Secure
- Never Trust User Input
- Defense in depth is the only defense
- Simple is easier to secure
- Peer review is critical to security

Secure Coding Principles (1 of 12)

List of security coding principles to prevent common security vulnerabilities

- Security through obscurity
- Secure the weakest link
- Use least privilege principle
- Secure by default
- Fail securely
- Apply defense in depth
- Do not trust user input
- Reduce attack surface
- Enable auditing and logging
- Keep security simple
- Separation of duties
- Fix security issues correctly
- Apply security in design phase

- Protect sensitive data
- Exception handling
- Secure memory management
- Protect memory or storage secrets
- Fundamentals of control granularity
- Fault tolerance
- Fault detection
- Fault removal
- Fault avoidance
- Loose coupling
- High cohesion
- Change management and version control

Secure Coding Principles (2 of 12)

Security Through Obscurity

- Security Through Obscurity (STO) relies on **preventing access** to certain users to protect internal data
- STO systems may have theoretical or actual security vulnerabilities, but designers believe that **flaws are unknown** and attackers are unlikely to find them
- Its usefulness has declined with the rise of **open systems**, networking, greater understanding of **programming techniques**, and increased capabilities of home users

Secure the Weakest Link

- Attackers target a system that is easy to **penetrate**
- For example, to gain access to the **encrypted** data on the network, attackers will not intercept the data and crack encryption; instead they will go after the **end points** of communication to find a **flaw** that discloses the data
- Identify and strengthen the **areas at risk** until levels of risk are satisfactory

Secure Coding Principles (3 of 12)

Use Least Privilege Principle

- Applications with maximum **system privileges** are vulnerable to the **attacks**
- **For example:** Many web applications use **database admin account** though not required to connect to the backend database, enhancing the impact of SQL injection exploits
- Protects application from malicious attacks by:
  - Determining and assigning only to those **privileges** required to complete the task
  - Avoiding applications that get **installed** and **run** by **default**
  - Writing applications that users having **non-administrative privileges** can actually use

Secure by Default

- The software solution or application provided to the users should be **security enabled** by default. If permitted, it is up to the user to reduce the security
- **For example:** By default the security feature **password aging** and **complexity** should be enabled



Secure Coding Principles (4 of 12)

Fail Securely

- The developer should not give application secrets through default **error messages**
- Application that discloses **confidential information** on failure assists attackers in creating an **attack**
- When an application fails, determine what may occur and ensure that it does not threaten the application
- Provide logical and useful **error messages** to the users and store the details in the **log file**

Apply Defense in Depth

- The architects and developers should consider all the **levels of the software** to impose security while developing software
- Implement security mechanisms at **different layers** that include network layer, kernel layer, physical layer, and the file system layer instead of same layer

Do not Trust User Input

- Protect the application from all **malicious inputs** coming from the user input to the application
- Consider all inputs as a malicious input and apply **security measures** to restrict them

Secure Coding Principles (5 of 12)

Reduce Attack Surface

- Application attack surface area is to be minimized by **reducing the number of entry points** into the application
- Remove or turn off the features, protocols, and functionality which are **not in use** in order to minimize number of vulnerabilities and the overall risk
- **For example:** If a vulnerability exists in a way an XML is parsed, denying XML from unknown users minimizes that security vulnerability

Enable Auditing and Logging

- Auditing and logging states how the **security related events** are recorded by an application
- Auditing enables **identification of attacks or intruders in progress**, whereas logging aids in identifying how an attack is performed
- Perform auditing and logging to gather information about **attacks**

Secure Coding Principles (6 of 12)

Keep Security Simple

- If the design is complex, it is hard to understand and **errors** are likely to occur in implementation, configuration, and use
- On the other hand, if the **complexity** of security mechanisms increases, the effort required to reach the appropriate level of **software assurance** also increases
- Avoid complex **architectures** and opt for **simpler approaches** that are fast and simple

Separation of Duties

- Separation of duties is the **key control of fraud**
- When assigning privileges, system roles are to be considered. In general system administrators are also the users as some super user privileges are required to make the system run
- For example: **System administrator** can set the password policy, turn off or on the system, etc. but should not be able to log in as a **super privileged** user

Secure Coding Principles (7 of 12)

Fix Security Issues Correctly

- When a security issue is identified, fix it, considering it as the actual problem, and then go through the **security process** as you do for the new code, ensuring that the fix does not introduce new errors
- **For example:** User capable of viewing **another user's account balance** by simply adjusting cookie. In this context once the security issue is fixed it has to be tested on all the applications as cookie handling code is shared among all applications

Apply Security in Design Phase

- Before starting the application development process, always consider **security issues** that can help prevent many **security vulnerabilities**
- Considering security issues helps you understand the **coding weaknesses** and **vulnerabilities** from the most obvious exploits

Protect Sensitive Data

- Do not **hard code** the sensitive data such as passwords in the program
- Use **data encryption mechanism** to transmit data over the network

Secure Coding Principles (8 of 12)

Exception Handling

- Events that disrupt the **coding process** are called exceptions
- Exception handling occurs when error conditions **interrupt** the normal flow of a program's execution
- Programmers have difficulty in designing for exception handling, as continuous checking for **error conditions** is necessary
- Proper use of exception handling helps to ensure proper **error handling**

Secure Memory Management

- Check memory bounds on the length of input variables, arrays, and arguments in order to prevent **buffer overflow attacks**
- Apply coding standards for simplicity, which help in **implementing security** in the **program** and keep things simple

Protect Memory or Storage Secrets

- **Encrypt secrets** to protect memory storage from ending up in crash dump file
- Use a perfect **cryptographic method** to perform encrypting secrets process
- Scrub secrets in memory storage before **deletion**

Secure Coding Principles (9 of 12)

Fundamentals of Control Granularity

- Applications managing sensitive data should possess **privacy protection** that can be achieved through flow control
- **Flow control** prevents data from non-secure information flows
- Systems with low-level granularity may lack control in **protecting data** from misuse
- System administrators are given better control of granularity at the URL level, shared directory level, and application level

- Different granularity is needed to control **information flows** within an application
- Role-based access control (RBAC) is a flow control model offering multiple levels of **control granularity**
  - Also called an n-leveled RBAC (LnRBAC), as a level of granularity is controlled by a level of RBAC control
  - Handles and solves problems caused by abnormal program stopping
  - Controls method invocation, write accesses, and avoids Trojan horses

Secure Coding Principles (10 of 12)

Fault Tolerance

- Strategy applied to **software design** (or system design) to permit system to continue functioning even in the presence of faults by enhancing its robustness

Fault Detection

- Closely linked to fault tolerance, used in **detecting faults** and producing appropriate responses of **system behavior**
- Examples include system monitors, safety monitors, built-in tests, loop-back tests, etc.

Fault Removal

- Removes faults during **design process**
- Examples include error detection, verification through inspection, built-in testing, correction functions, etc.

Fault Avoidance

- Avoids errors that contribute to system faults during the **development process**
- Examples include defensive programming, error minimization during design process, minimization of safety critical code, using appropriate SDLC techniques, etc.

Secure Coding Principles (11 of 12)

Loose Coupling

- Procedures that **operate independently** from other procedures are called loosely coupled procedures
- Loose coupling describes the **exchange relationship** that occurs between two or more systems or organizations
- The concept loose coupling of systems is adopted when either source or destination machines are supposed to be **changed frequently**
- Loose coupling can be achieved in **web services** or **service-oriented architecture** by hiding the implementation details from the caller

Well-known techniques that create loose coupling

- Vendor and platform independent messages
- Coarse-grained, self-describing, and self-contained messages
- Well-defined interfaces
- Extensible versionable interfaces
- Constrained interfaces
- Stateless messaging
- Human readable strings (URIs) for service and instance addresses
- Stateless messaging where possible and appropriate
- Humans controlling clients where possible and appropriate
- Asynchronous exchange patterns where possible and appropriate

Secure Coding Principles (12 of 12)

High Cohesion

- Procedures that perform a single function
- Cohesion is a measure of identifying the extent of strongly related functionalities in the **source code** of the single module
- A particular class is said to have high cohesion if the methods in that class perform **similar actions** in many aspects
- This high cohesively code increases the code **readability** and **reusability** without increasing complexity
- Procedures that are more reliable, easy to read, and maintain are usually **loosely coupled** and have **high cohesion**

Change Management and Version Control

- **Integrity** of system changes is controlled by a well-defined process called **configuration management**
- Allows all **stakeholders** to know upcoming changes
- Change management controls integration **costs** and **development**
- Required changes to code may cause **vulnerabilities**
- Anticipation of change requirements may **limit negative effects**

Guidelines for Developing Secure Codes (1 of 8)

Secure Coding Guidelines

Validate All User's Input

- Never believe that the input from HTML forms is valid
- The user's input must contain valid characters and valid values before it is processed by the application
- Be restrictive and check all the input data at multiple levels

Sanitize Input Values

- Incoming data may contain malicious codes and invalid strings that may cause buffer over-runs, integer arithmetic errors, SQL injection, and CSS attacks
- Use regular expressions to stop untrusted incoming data
- Disallow all special characters when validating the input
- Implement parameter substitutions technique in the database interface rather writing a query strings using concatenation

Guidelines for Developing Secure Codes (2 of 8)

Use the Principle of Least Privilege

- Avoid running the applications with **administrative privileges** that are not needed
- Protect the applications from **malicious attacks**, certainly reducing the damage to the code
- Avoid the applications that get **installed and run by default**
- Write applications that users with **non-administrative privileges** can actually use

Handle Sensitive Security Information with Care

- Store **sensitive information** in appropriate and more secure locations
- User's personal information should not be written in any application or system log files, **as logs are readable**
- Always encrypt the sensitive data stored on the **mass storage**
- To handle passwords, use **hash keys**

Guidelines for Developing Secure Codes (3 of 8)

Scan Existing Applications for Code Weaknesses

- Always scan the existing applications to know **exploitable weaknesses** regularly
- Always look for updates to **patch** the operating system and secure the **TCP/IP stack** on which the existing applications run
- Always make sure the existing applications and programs use **secure file** permissions

Use Threat Models

- Threat modeling helps to analyze the **potential risks** of the software and also helps in writing more **secure code**
- Threat models help to prove the **authenticity**, source destination (remote or local) of the **incoming data**, and tell the nature of the data that is being protected
- Always use accurate and up-to-date threat models to check the **code accuracy**

Perform Software Testing

- Deploy a **testing risk-based testing methodology** based on both the system's architectural reality and the attacker's mindset to gauge software security adequately
- Use **fuzzing testing technique** and mainly target network protocols and file formats
- Fuzz all codes by using a file fuzzer for **untrusted** or **random** data that could be an input to the program

Guidelines for Developing Secure Codes (4 of 8)

- Use **application environment tools** to do input validation
- Do not provide **hints** to attackers
- Do not add **comments** telling what the code does
- Compile the code and eliminate the **warnings** by modifying the code
- Identify and eliminate additional **security flaws** in the code using static and dynamic analysis tools
- Build a **secure software architecture** and design software to enforce security policies
- Keep the design **as simple as possible**, as errors are likely to occur if the design is complex
- By default **restrict access** to users

Guidelines for Developing Secure Codes (5 of 8)

Restrict access to users unless it is necessary to permit access based on certain conditions	Ensure that the program executes with least set of **privileges** required to complete the task
Sanitize the data passed to **complex subsystems** as it may contain special characters that activate commands causing software vulnerability	Implement **parameter substitutions technique** in the database interface rather than writing a query strings using concatenation
Do not rely on single layer of security instead use **multiple layers of security**; to keep attackers away	Implement **quality assurance techniques** such as penetration testing, fuzz testing, code reviews, and audits etc. to identify and eliminate vulnerabilities
Implement **secure coding standard** for platform and language development	Perform **threat modeling** to mitigate potential threats to the software

Guidelines for Developing Secure Codes (6 of 8)

- Use version control to track changes made to the code; this saves the development time in case of flaw
- Never build SQL statements using string replacement or concatenation. Use parameterized SQL statements
- Use latest compilers that provide security against coding errors

- Develop code with proper exception/error handling
- Do not develop code using the functions that are prohibited

- Implement cryptography techniques to encrypt the sensitive information
- Regularly update the secure coding checklist based on new software vulnerabilities/bugs arrived
- Be familiar with all the secure coding principles and strategies to mitigate common threats

Guidelines for Developing Secure Codes (7 of 8)

Do not include **confidential information** such as login credentials in the program's source code

Ensure that programs will self limit their own **resource consumption**

Temporary data such as **cache** should be deleted ideally and immediately after usage by the program

Ensure that the software solution/application developed should be **security enabled** by default

Ensure that program fails securely without revealing any **sensitive data** accessible to exploit vulnerabilities

Validate all **parameters** of public and exported application programming interfaces

Error messages should be **user friendly** and display the fact error occurred and not the cause or nature of the error

Avoid hard-coding passwords that may lead to **authentication failure**

Guidelines for Developing Secure Codes (8 of 8)

- Never read a file that is still being written by another process
- Take care of temporary files as they can be altered if they do not have any secure permissions
- Do not use insecure system calls and switches as they make a way for unauthorized users to exploit vulnerabilities
- Perform bounds checking for input variables, arrays, and arguments to prevent buffer overflows attacks
- Safeguard weak file and group permissions policies
- Always look for updates to patch the OS and secure the TCP/IP stack on which the existing applications run
- Do not invoke system command line/shell from within the application program
- Scan the existing applications to know exploitable weaknesses regularly

Bonus

Please Give Comment

Lab

- Execute Reflected XSS Attack
- Exploit Information Disclosure Vulnerability
- Execute a code injection Attack, Exploiting File upload Vulnerability
- Execute SQL Injection Attack

Lab Answer I

- Execute Reflected XSS Attack
  - open browser
  - Click Products Button and Purchase these product
    - Go to Shopping cart
    - Inject Using javascript alert
- Exploit Information Disclosure Vulnerability
  - Open Browser
  - enter url with random page name (query String)

Lab Answer II

- Execute a code injection Attack, Exploiting File upload Vulnerability
  - Open Browser
  - Click Contact us
  - Upload file info.php
  - enter Url with page ../uploads/info
- Execute SQL Injection Attack
  - Open Browser
  - Click Our member
  - Type : a%' UNION SELECT `password`, `password`, `password`, `password` FROM `members`;–