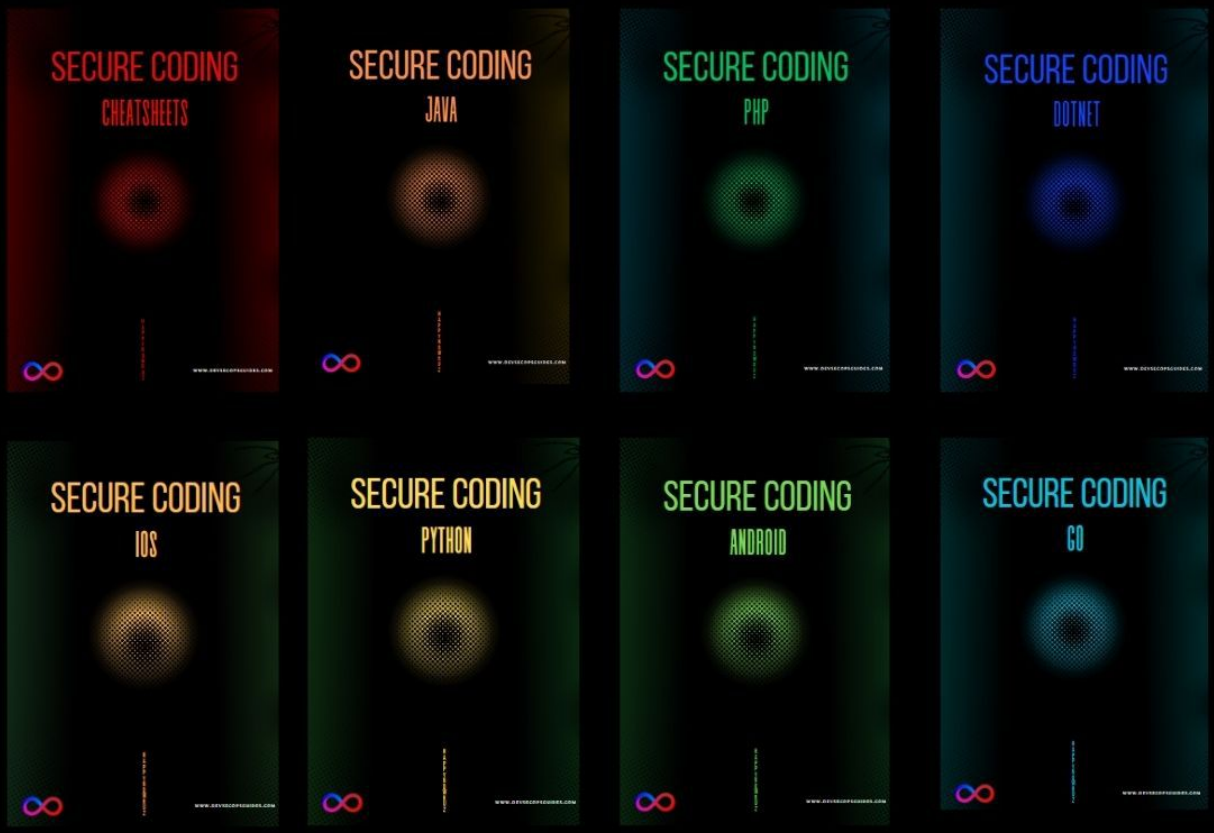
SECURE CODING

CHEATSHEETS

HAPPYNOWRUZ

WWW.DEVSECOPSGUIDES.COM

BOOKS



LABS

portswigger | secure code warrior | veracode


CYRUS THE GREAT:
GOOD THOUGHTS, GOOD WORDS, GOOD DEEDS

TABLE OF CONTENT

Secure Coding Cheatsheets

· Mar 22, 2024 · 41 min read

Table of contents

- Security Misconfiguration
  - Non-Compliant Code with Vulnerability:
    - Description of Vulnerability:
  - Compliant Code with Mitigation:
    - Description of Mitigation:
- Vulnerable and Outdated Components
  - Non-Compliant Code with Vulnerability:
    - Description of Vulnerability:
  - Compliant Code with Mitigation:
    - Description of Mitigation:
- Identification and Authentication Failures
  - Non-Compliant Code with Vulnerability:
    - Description of Vulnerability:
  - Compliant Code with Mitigation:
    - Description of Mitigation:
- Software and Data Integrity Failures
  - Non-Compliant Code with Vulnerability:
    - Description of Vulnerability:
  - Compliant Code with Mitigation:
    - Description of Mitigation:
- Security Logging and Monitoring Failures
  - Non-Compliant Code with Vulnerability:
    - Description of Vulnerability:
  - Compliant Code with Mitigation:
    - Description of Mitigation:
- Server-Side Request Forgery
  - Non-Compliant Code with Vulnerability:
    - Description of Vulnerability:

- Compliant Code with Mitigation:
  - Description of Mitigation:
- PHP
  - 1. Broken Access Control:
    - Non-Compliant Code:
    - Description:
    - Compliant Code:
  - 2. Cryptographic Failures:
    - Non-Compliant Code:
    - Description:
    - Compliant Code:
  - 3. Injection:
    - Non-Compliant Code:
    - Description:
    - Compliant Code:
  - 4. Insecure Design:
    - Non-Compliant Code:
    - Description:
    - Compliant Code:
  - 5. Security Misconfiguration:
    - Non-Compliant Code:
    - Description:
    - Compliant Code:
  - 6. Vulnerable and Outdated Components:
    - Non-Compliant Code:
    - Description:
    - Compliant Code:
- Go

- 1. Broken Access Control:
  - Non-Compliant Code:
  - Description:
  - Compliant Code:
  - Description:
- 2. Cryptographic Failures:
  - Non-Compliant Code:
  - Description:
  - Compliant Code:
  - Description:
- 3. Injection:
  - Non-Compliant Code:
  - Description:
  - Compliant Code:
  - Description:

- Python
  - 1. Broken Access Control:
    - Non-Compliant Code:
    - Description:
    - Compliant Code:
    - Description:
  - 2. Cryptographic Failures:
    - Non-Compliant Code:
    - Description:
    - Compliant Code:
    - Description:
  - 3. Injection:
    - Non-Compliant Code:

- Description:
- Compliant Code:
- Description:
- 4. Insecure Design:
  - Non-Compliant Code:
  - Description:
  - Compliant Code:
  - Description:

- Java
  - 1. Broken Access Control:
    - Non-Compliant Code:
    - Description:
    - Compliant Code:
    - Description:
  - 2. Cryptographic Failures:
    - Non-Compliant Code:
    - Description:
    - Compliant Code:
    - Description:
  - 3. Injection:
    - Non-Compliant Code:
    - Description:
    - Compliant Code:
    - Description:

- Android Java
  - 1. Improper Credential Usage:
    - Non-Compliant Code:
    - Description:

- Compliant Code:
- Description:
- 2. Inadequate Supply Chain Security:
  - Non-Compliant Code:
  - Description:
  - Compliant Code:
  - Description:
- 3. Insecure Authentication/Authorization:
  - Non-Compliant Code:
  - Description:
  - Compliant Code:
  - Description:
- 4. Insufficient Input/Output Validation:
  - Non-Compliant Code:
  - Description:
  - Compliant Code:
  - Description:
- 5. Insecure Communication:
  - Non-Compliant Code:
  - Description:
  - Compliant Code:
  - Description:
- IOS Swift
  - 1. Improper Credential Usage:
    - Non-Compliant Code:
    - Description:
    - Compliant Code:
    - Description:

- 2. Inadequate Supply Chain Security:
  - Non-Compliant Code:
  - Description:
  - Compliant Code:
  - Description:
- 3. Insecure Authentication/Authorization:
  - Non-Compliant Code:
  - Description:
  - Compliant Code:
  - Description:
- 4. Insufficient Input/Output Validation:
  - Non-Compliant Code:
  - Description:
  - Compliant Code:
  - Description:
- 5. Insecure Communication:
  - Non-Compliant Code:
  - Description:
  - Compliant Code:
  - Description:
- 6. Inadequate Privacy Controls:
  - Non-Compliant Code:
  - Description:
  - Compliant Code:
  - Description:

Show less

In today's interconnected digital landscape, security is paramount for developers across various platforms and programming languages. With cyber threats evolving rapidly, ensuring the security of software applications has become more critical than ever. From mobile applications to web services and enterprise solutions, developers must adhere to secure coding practices to mitigate the risk of data breaches, unauthorized access, and other security vulnerabilities.

Secure coding cheatsheets serve as invaluable resources for developers, offering concise yet comprehensive guidance on secure coding practices specific to different programming languages and platforms. In this article, we delve into the world of secure coding cheatsheets, exploring best practices and recommendations tailored to popular programming languages and frameworks, including Android, iOS, PHP, .NET, Python, Go, and Java.

Each section of this article will provide an overview of secure coding considerations for the respective platform or language, highlighting common security vulnerabilities and offering practical tips and guidelines for writing secure code. Whether you're a seasoned developer or just starting your journey in software development, these cheatsheets will empower you to write more secure and resilient code, ultimately enhancing the security posture of your applications.

Dotnet

broken access control

Non-Compliant Code with Vulnerability:

COPY

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;

namespace BrokenAccessControl.Controllers
```

```
{
    public class UserController : Controller
    {
        // Non-compliant code allowing unauthorized access to
sensitive information
        public ActionResult ViewSensitiveData()
        {
            // Assume that this method should only be accessible by
certain privileged users
            string sensitiveData = FetchSensitiveDataFromDatabase();

            return View("SensitiveDataView", sensitiveData);
        }

        // Method to fetch sensitive data from the database
        private string FetchSensitiveDataFromDatabase()
        {
            // Insecure implementation, not checking user's
authorization
            // Vulnerability: Lack of proper access control, allowing
any authenticated user to access sensitive data.
            return "This is sensitive data from the database.";
        }
    }
}
```

Description of Vulnerability:

The non-compliant code provided above contains a broken access control vulnerability. The `ViewSensitiveData` action method does not implement any checks to ensure that only authorized users can access sensitive information. As a result, any authenticated user can view the sensitive data by accessing this endpoint. This vulnerability can lead to unauthorized access to sensitive data, violating the confidentiality of the application's data.

Compliant Code with Mitigation:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;

namespace BrokenAccessControl.Controllers
{
    public class UserController : Controller
    {
        // Compliant code ensuring proper access control
        [Authorize(Roles = "Admin")] // Restrict access to users in
the "Admin" role
        public ActionResult ViewSensitiveData()
        {
            string sensitiveData = FetchSensitiveDataFromDatabase();
            return View("SensitiveDataView", sensitiveData);
        }

        // Method to fetch sensitive data from the database
        private string FetchSensitiveDataFromDatabase()
        {
            // Secure implementation, sensitive data retrieval is
conditional on user's authorization
            return "This is sensitive data from the database.";
        }
    }
}
```

Description of Mitigation:

The compliant code provided above addresses the broken access control vulnerability by implementing proper access control checks. The `[Authorize(Roles = "Admin")]` attribute is applied to the `ViewSensitiveData` action method, restricting access to only those users who are in the "Admin" role. This ensures that sensitive data can only be accessed by authorized users, mitigating the risk of unauthorized

access. By enforcing access control at the application level, the confidentiality of sensitive data is preserved, reducing the likelihood of security breaches.

Cryptographic Failures

Non-Compliant Code with Vulnerability:

COPY

```
using System;
using System.Security.Cryptography;
using System.Text;

namespace CryptographicFailures
{
    public class CryptoService
    {
        // Non-compliant code with cryptographic vulnerability
        public string Encrypt(string data, string key)
        {
            byte[] keyBytes = Encoding.UTF8.GetBytes(key);
            byte[] dataBytes = Encoding.UTF8.GetBytes(data);

            using (var aes = Aes.Create())
            {
                aes.Key = keyBytes;
                aes.Mode = CipherMode.ECB; // Vulnerable mode
                aes.Padding = PaddingMode.PKCS7;

                using (var encryptor = aes.CreateEncryptor())
                {
                    byte[] encryptedData =
encryptor.TransformFinalBlock(dataBytes, 0, dataBytes.Length);
                    return Convert.ToBase64String(encryptedData);
                }
            }
        }
```

```
        // Non-compliant code with cryptographic vulnerability
        public string Decrypt(string encryptedData, string key)
        {
            byte[] keyBytes = Encoding.UTF8.GetBytes(key);
            byte[] encryptedBytes =
Convert.FromBase64String(encryptedData);

            using (var aes = Aes.Create())
            {
                aes.Key = keyBytes;
                aes.Mode = CipherMode.ECB; // Vulnerable mode
                aes.Padding = PaddingMode.PKCS7;

                using (var decryptor = aes.CreateDecryptor())
                {
                    byte[] decryptedData =
decryptor.TransformFinalBlock(encryptedBytes, 0,
encryptedBytes.Length);
                    return Encoding.UTF8.GetString(decryptedData);
                }
            }
        }
    }
}
```

Description of Vulnerability:

The non-compliant code provided above contains cryptographic vulnerabilities due to the use of the vulnerable cipher mode `CipherMode.ECB`. ECB (Electronic Codebook) mode is vulnerable to known plaintext attacks and does not provide semantic security, as identical plaintext blocks are encrypted into identical ciphertext blocks. This vulnerability can lead to various cryptographic attacks, including plaintext recovery and pattern recognition.

Compliant Code with Mitigation:

```
using System;
using System.Security.Cryptography;
using System.Text;

namespace CryptographicFailures
{
    public class CryptoService
    {
        // Compliant code with proper cryptographic implementation
        public string Encrypt(string data, string key)
        {
            byte[] keyBytes = Encoding.UTF8.GetBytes(key);
            byte[] dataBytes = Encoding.UTF8.GetBytes(data);

            using (var aes = Aes.Create())
            {
                aes.Key = keyBytes;
                aes.Mode = CipherMode.CBC; // Using CBC mode for
better security
                aes.Padding = PaddingMode.PKCS7;

                aes.GenerateIV(); // Generate a random IV for each
encryption

                using (var encryptor = aes.CreateEncryptor())
                {
                    byte[] encryptedData =
encryptor.TransformFinalBlock(dataBytes, 0, dataBytes.Length);
                    return
Convert.ToBase64String(aes.IV.Concat(encryptedData).ToArray());
                }
            }
        }

        // Compliant code with proper cryptographic implementation
        public string Decrypt(string encryptedData, string key)
        {
            byte[] keyBytes = Encoding.UTF8.GetBytes(key);
```

```
            byte[] encryptedBytesWithIV =
Convert.FromBase64String(encryptedData);

            using (var aes = Aes.Create())
            {
                aes.Key = keyBytes;
                aes.Mode = CipherMode.CBC; // Using CBC mode for
decryption
                aes.Padding = PaddingMode.PKCS7;

                // Extract IV from the encrypted data
                byte[] iv = encryptedBytesWithIV.Take(aes.BlockSize /
8).ToArray();
                byte[] encryptedBytes =
encryptedBytesWithIV.Skip(aes.BlockSize / 8).ToArray();

                aes.IV = iv; // Set IV for decryption

                using (var decryptor = aes.CreateDecryptor())
                {
                    byte[] decryptedData =
decryptor.TransformFinalBlock(encryptedBytes, 0,
encryptedBytes.Length);
                    return Encoding.UTF8.GetString(decryptedData);
                }
            }
        }
    }
}
```

Description of Mitigation:

The compliant code provided above addresses cryptographic vulnerabilities by using the CBC (Cipher Block Chaining) mode for encryption and decryption, which provides better security compared to ECB mode. CBC mode XORs each plaintext block with the previous ciphertext block before encryption, preventing identical plaintext blocks from encrypting into identical ciphertext blocks. Additionally, a unique IV (Initialization Vector) is generated for each encryption operation and included with the ciphertext

to ensure randomness and prevent patterns. By implementing proper cryptographic primitives and modes, the security of the application's cryptographic operations is enhanced, reducing the risk of cryptographic attacks.

Injection

Non-Compliant Code with Vulnerability:

COPY

```
using System;
using System.Data.SqlClient;

namespace InjectionVulnerabilities
{
    public class DatabaseService
    {
        // Non-compliant code with SQL injection vulnerability
        public bool AuthenticateUser(string username, string password)
        {
            string query = $"SELECT COUNT(*) FROM Users WHERE Username
= '{username}' AND Password = '{password}'";

            using (var connection = new
SqlConnection("connection_string_here"))
            using (var command = new SqlCommand(query, connection))
            {
                connection.Open();
                int count = (int)command.ExecuteScalar();
                return count > 0;
            }
        }
    }
}
```

Description of Vulnerability:

The non-compliant code provided above contains a SQL injection vulnerability. The SQL query is constructed by directly interpolating user inputs (`username` and `password`) into the query string. An attacker can exploit this vulnerability by providing malicious inputs that alter the structure of the SQL query, potentially leading to unauthorized access, data leakage, or database manipulation. For example, an attacker could input `' OR '1'='1` as the password, effectively bypassing authentication.

Compliant Code with Mitigation:

COPY

```
using System;
using System.Data.SqlClient;

namespace InjectionVulnerabilities
{
    public class DatabaseService
    {
        // Compliant code with parameterized query to prevent SQL
injection
        public bool AuthenticateUser(string username, string password)
        {
            string query = "SELECT COUNT(*) FROM Users WHERE Username
= @Username AND Password = @Password";

            using (var connection = new
SqlConnection("connection_string_here"))
            using (var command = new SqlCommand(query, connection))
            {
                command.Parameters.AddWithValue("@Username",
username);
                command.Parameters.AddWithValue("@Password",
password);

                connection.Open();
                int count = (int)command.ExecuteScalar();
                return count > 0;
            }
```

```
        }
    }
}
```

Description of Mitigation:

The compliant code provided above addresses the SQL injection vulnerability by using parameterized queries. In a parameterized query, placeholders (e.g., `@Username` and `@Password`) are used instead of directly concatenating user inputs into the SQL query string. User inputs are then supplied as parameters to the query, ensuring that they are treated as data rather than executable code. This prevents attackers from injecting malicious SQL code into the query, thus mitigating the risk of SQL injection attacks. Parameterized queries are a recommended best practice for preventing injection vulnerabilities in database interactions.

Insecure Design

Non-Compliant Code with Vulnerability:

COPY

```
using System;

namespace InsecureDesign
{
    public class PaymentProcessor
    {
        // Non-compliant code with insecure design vulnerability
        public void ProcessPayment(string creditCardNumber, decimal
amount)
        {
            // Assume credit card processing logic here
            Console.WriteLine($"Processing payment of {amount} with
credit card number {creditCardNumber}");
        }
    }
```

```
    public class UserController
    {
        private readonly PaymentProcessor _paymentProcessor;

        public UserController()
        {
            _paymentProcessor = new PaymentProcessor();
        }

        public void MakePayment(string creditCardNumber, decimal
amount)
        {
            // Assume user authentication and authorization checks
here

            // Insecure design: Passing sensitive credit card
information as method arguments
            _paymentProcessor.ProcessPayment(creditCardNumber,
amount);
        }
    }
}
```

Description of Vulnerability:

The non-compliant code provided above contains an insecure design vulnerability related to passing sensitive data (in this case, credit card numbers) as method arguments. This violates the principle of least privilege and can expose sensitive information to unauthorized access or interception. If an attacker gains access to the memory or logs of the application, they could potentially capture sensitive credit card information, leading to data breaches and financial losses.

Compliant Code with Mitigation:

COPY

```
using System;
```

```
namespace InsecureDesign
{
    public class PaymentProcessor
    {
        // Compliant code addressing insecure design vulnerability
        public void ProcessPayment(PaymentInfo paymentInfo)
        {
            // Assume credit card processing logic here
            Console.WriteLine($"Processing payment of
{paymentInfo.Amount} with credit card number
{paymentInfo.MaskedCreditCardNumber}");
        }
    }

    public class PaymentInfo
    {
        public string MaskedCreditCardNumber { get; set; }
        public decimal Amount { get; set; }
        // Additional properties related to payment information can be
added here
    }

    public class UserController
    {
        private readonly PaymentProcessor _paymentProcessor;

        public UserController()
        {
            _paymentProcessor = new PaymentProcessor();
        }

        public void MakePayment(PaymentInfo paymentInfo)
        {
            // Assume user authentication and authorization checks
here

            // Passing payment information encapsulated in a class
instance
            _paymentProcessor.ProcessPayment(paymentInfo);
        }
```

```
    }
}
```

Description of Mitigation:

The compliant code provided above addresses the insecure design vulnerability by encapsulating sensitive payment information (such as credit card numbers) into a separate class (`PaymentInfo`) rather than passing them directly as method arguments. By encapsulating sensitive data, we reduce the exposure of this information and improve the overall security of the application. Additionally, by passing a single object containing all necessary payment information, we adhere to the principle of least privilege, ensuring that only the necessary data is accessed by each component of the system. This mitigates the risk of unauthorized access or interception of sensitive information, enhancing the security posture of the application.

Security Misconfiguration

Non-Compliant Code with Vulnerability:

COPY

```
using System;
using System.Web.Mvc;

namespace SecurityMisconfiguration
{
    public class UserController : Controller
    {
        // Non-compliant code with security misconfiguration
vulnerability
        public ActionResult Index()
        {
            // Assume this action method returns sensitive information
without proper access control
            string sensitiveData = "This is sensitive data.";
            return View("Index", sensitiveData);
        }
```

```
    }
}
```

Description of Vulnerability:

The non-compliant code provided above contains a security misconfiguration vulnerability. In this example, the action method `Index` returns sensitive information (`sensitiveData`) without applying proper access controls. This could allow any user, including unauthenticated or unauthorized users, to access and view sensitive data. Security misconfigurations such as this can lead to unauthorized access, data breaches, and compromise of sensitive information.

Compliant Code with Mitigation:

COPY

```
using System;
using System.Web.Mvc;

namespace SecurityMisconfiguration
{
    public class UserController : Controller
    {
        // Compliant code addressing security misconfiguration
vulnerability
        [Authorize] // Apply authorization to restrict access to
authenticated users
        public ActionResult Index()
        {
            // Assume this action method returns sensitive information
            string sensitiveData = "This is sensitive data.";
            return View("Index", sensitiveData);
        }
    }
}
```

Description of Mitigation:

```
    }
}
```

Description of Vulnerability:

The non-compliant code provided above contains a security misconfiguration vulnerability. In this example, the action method `Index` returns sensitive information (`sensitiveData`) without applying proper access controls. This could allow any user, including unauthenticated or unauthorized users, to access and view sensitive data. Security misconfigurations such as this can lead to unauthorized access, data breaches, and compromise of sensitive information.

Compliant Code with Mitigation:

COPY

```
using System;
using System.Web.Mvc;

namespace SecurityMisconfiguration
{
    public class UserController : Controller
    {
        // Compliant code addressing security misconfiguration
vulnerability
        [Authorize] // Apply authorization to restrict access to
authenticated users
        public ActionResult Index()
        {
            // Assume this action method returns sensitive information
            string sensitiveData = "This is sensitive data.";
            return View("Index", sensitiveData);
        }
    }
}
```

Description of Mitigation:

The compliant code provided above addresses the security misconfiguration vulnerability by applying proper access controls. The `[Authorize]` attribute is added to the `Index` action method, ensuring that only authenticated users can access the sensitive data. By enforcing authentication, we mitigate the risk of unauthorized access to sensitive information. Additionally, other security measures such as role-based access control (RBAC) or further authorization checks can be implemented as needed to restrict access to specific user roles or permissions. This helps in maintaining the confidentiality, integrity, and availability of sensitive data within the application.

Vulnerable and Outdated Components

Non-Compliant Code with Vulnerability:

COPY

```
using System;
using Newtonsoft.Json;

namespace VulnerableComponents
{
    public class UserController
    {
        // Non-compliant code using outdated and vulnerable JSON.NET
library
        public void ProcessJson(string json)
        {
            // Assume processing JSON data
            var deserializedData =
JsonConvert.DeserializeObject(json);

            // Perform further operations with deserialized data
        }
    }
}
```

Description of Vulnerability:

The non-compliant code provided above contains a vulnerability related to the use of an outdated and potentially vulnerable library - `Newtonsoft.Json`. This library is commonly used for JSON serialization and deserialization in .NET applications. However, using outdated versions of this library can expose the application to known security vulnerabilities, such as deserialization attacks. Attackers can exploit these vulnerabilities to execute arbitrary code, perform unauthorized actions, or achieve remote code execution. Using outdated components without patching or updating exposes the application to unnecessary security risks.

Compliant Code with Mitigation:

COPY

```
using System;
using System.Text.Json;

namespace SecureComponents
{
    public class UserController
    {
        // Compliant code using up-to-date and secure JSON library
        public void ProcessJson(string json)
        {
            // Use the modern and secure System.Text.Json library for
JSON processing
            var deserializedData = JsonSerializer.Deserialize<object>
(json);

            // Perform further operations with deserialized data
        }
    }
}
```

Description of Mitigation:

The compliant code provided above addresses the vulnerability related to vulnerable and outdated components by using an up-to-date and secure alternative - `System.Text.Json`. This library is included in the .NET framework and provides JSON serialization and deserialization functionality. Unlike the outdated Newtonsoft.Json library, System.Text.Json is regularly maintained and updated by Microsoft, with a focus on security and performance. By using modern and secure components, we mitigate the risk of known vulnerabilities and ensure the overall security of the application. Additionally, it's important to regularly monitor for updates and patches to libraries and dependencies used in the application to address any newly discovered vulnerabilities promptly.

Identification and Authentication Failures

Non-Compliant Code with Vulnerability:

COPY

```
using System;
using System.Web.Mvc;

namespace AuthenticationFailures
{
    public class UserController : Controller
    {
        // Non-compliant code with authentication failure
vulnerability
        public ActionResult Login(string username, string password)
        {
            // Assume authentication logic here
            if (username == "admin" && password == "admin123")
            {
                // Successful authentication, redirect to dashboard
                return RedirectToAction("Dashboard", "Home");
            }
            else
            {
                // Authentication failed, return error view
```

```
                return View("Error");
            }
        }
    }
}
```

Description of Vulnerability:

The non-compliant code provided above contains an authentication failure vulnerability. In this example, authentication is based solely on comparing the username and password provided by the user with hardcoded values (`admin` and `admin123`). This approach is insecure as it lacks proper authentication mechanisms such as password hashing, salting, or user authentication against a secure data store. Hardcoded credentials make it easier for attackers to guess or brute-force login credentials, leading to unauthorized access to the application.

Compliant Code with Mitigation:

COPY

```
using System;
using System.Web.Mvc;

namespace AuthenticationFailures
{
    public class UserController : Controller
    {
        // Compliant code addressing authentication failure
vulnerability
        public ActionResult Login(string username, string password)
        {
            // Authenticate user against a secure data store (e.g.,
database)
            if (IsValidUser(username, password))
            {
                // Successful authentication, redirect to dashboard
                return RedirectToAction("Dashboard", "Home");
            }
```

```
            else
            {
                // Authentication failed, return error view
                return View("Error");
            }
        }

        // Method to validate user credentials against a secure data
store
        private bool IsValidUser(string username, string password)
        {
            // Logic to authenticate user against a secure data store
(e.g., database)
            // Return true if user credentials are valid, otherwise
return false
            // Implement secure password hashing and salting
mechanisms for storing and comparing passwords
            // Example implementation:
            // return UserRepository.ValidateCredentials(username,
password);
            return false; // Placeholder return value
        }
    }
}
```

Description of Mitigation:

The compliant code provided above addresses the authentication failure vulnerability by implementing proper authentication mechanisms. Instead of comparing user credentials against hardcoded values, the `Login` action method calls the `IsValidUser` method to authenticate the user against a secure data store (e.g., database). This allows for the implementation of secure authentication features such as password hashing, salting, and secure storage of user credentials. By validating user credentials against a secure data store, we enhance the security of the authentication process and reduce the risk of unauthorized access to the application. Additionally, using secure authentication mechanisms makes it more difficult for attackers to compromise user accounts through brute-force attacks or credential guessing.

Software and Data Integrity Failures

Non-Compliant Code with Vulnerability:

COPY

```
using System;
using System.IO;

namespace DataIntegrityFailures
{
    public class DataManipulator
    {
        // Non-compliant code with data integrity vulnerability
        public void WriteDataToFile(string filePath, string data)
        {
            // Write data to the specified file
            File.WriteAllText(filePath, data);
        }

        // Non-compliant code with software integrity vulnerability
        public void LoadAssembly(string assemblyPath)
        {
            // Load and execute an assembly from the specified path
            var assembly =
System.Reflection.Assembly.LoadFrom(assemblyPath);
            assembly.EntryPoint.Invoke(null, null);
        }
    }
}
```

Description of Vulnerability:

The non-compliant code provided above contains vulnerabilities related to both software and data integrity. In the `WriteDataToFile` method, data is directly written to a file using the `File.WriteAllText` method. This approach lacks integrity checks, such as hashing or digital signatures, which can ensure that the data has not been tampered with during transmission or storage. Without integrity checks, attackers

could modify the data stored in the file, leading to potential data corruption or unauthorized modifications.

Similarly, in the `LoadAssembly` method, an assembly is loaded and executed from the specified path without performing any integrity checks. This can lead to software integrity failures if the loaded assembly has been tampered with or replaced by a malicious version. Attackers could exploit this vulnerability to execute arbitrary code, leading to system compromise or unauthorized actions.

Compliant Code with Mitigation:

COPY

```
using System;
using System.IO;
using System.Security.Cryptography;

namespace SecureDataIntegrity
{
    public class DataManipulator
    {
        // Compliant code addressing data integrity vulnerability
        public void WriteDataToFile(string filePath, string data)
        {
            // Calculate the SHA-256 hash of the data
            byte[] dataBytes =
System.Text.Encoding.UTF8.GetBytes(data);
            byte[] hashBytes;
            using (SHA256 sha256 = SHA256.Create())
            {
                hashBytes = sha256.ComputeHash(dataBytes);
            }
            string hash = BitConverter.ToString(hashBytes).Replace("-
", string.Empty);

            // Write the data and its hash to the file
            File.WriteAllText(filePath, $"{data}\nHash: {hash}");
        }
```

```
        // Compliant code addressing software integrity vulnerability
        public void LoadAssembly(string assemblyPath, string
expectedHash)
        {
            // Read the assembly bytes from the file
            byte[] assemblyBytes = File.ReadAllBytes(assemblyPath);

            // Calculate the SHA-256 hash of the assembly
            byte[] hashBytes;
            using (SHA256 sha256 = SHA256.Create())
            {
                hashBytes = sha256.ComputeHash(assemblyBytes);
            }
            string actualHash =
BitConverter.ToString(hashBytes).Replace("-", string.Empty);

            // Compare the actual hash with the expected hash
            if (actualHash.Equals(expectedHash,
StringComparison.OrdinalIgnoreCase))
            {
                // Load and execute the assembly if the hashes match
                var assembly =
System.Reflection.Assembly.Load(assemblyBytes);
                assembly.EntryPoint.Invoke(null, null);
            }
            else
            {
                // Log or handle the integrity check failure
                Console.WriteLine("Integrity check failed. The
assembly has been tampered with.");
            }
        }
    }
}
```

Description of Mitigation:

The compliant code provided above addresses both software and data integrity vulnerabilities by implementing integrity checks.

In the `WriteDataToFile` method, before writing data to the file, a SHA-256 hash of the data is calculated and stored alongside the data in the file. This hash acts as a checksum, allowing the receiver to verify the integrity of the data. If the data is modified, the hash will no longer match, indicating data tampering.

In the `LoadAssembly` method, before loading and executing the assembly, the SHA-256 hash of the assembly is calculated and compared with the expected hash. If the hashes match, the assembly is considered intact, and it is safe to execute. If the hashes do not match, it indicates that the assembly has been tampered with, and appropriate action can be taken (e.g., logging the integrity check failure). These integrity checks help ensure that both data and software remain unaltered and trustworthy, mitigating the risk of unauthorized modifications or execution of malicious code.

Security Logging and Monitoring Failures

Non-Compliant Code with Vulnerability:

COPY

```
using System;
using System.IO;

namespace SecurityLoggingFailures
{
    public class UserController
    {
        // Non-compliant code with security logging failure
        public void Login(string username, string password)
        {
            // Assume authentication logic here
            if (username == "admin" && password == "admin123")
            {
                // Successful login
                Console.WriteLine($"User '{username}' logged in
successfully.");
            }
```

```
            else
            {
                // Failed login
                Console.WriteLine($"Failed login attempt for user
'{username}'.");
            }
        }
    }
}
```

Description of Vulnerability:

The non-compliant code provided above contains a security logging failure. While it logs login attempts (both successful and failed) using `Console.WriteLine`, this approach is inadequate for effective security logging and monitoring. Console output is not a robust logging mechanism and lacks features such as timestamping, log severity levels, and centralized log management. Without proper logging, security incidents may go unnoticed, making it challenging to detect and respond to unauthorized access attempts or suspicious activities effectively.

Compliant Code with Mitigation:

COPY

```
using System;
using System.IO;

namespace SecurityLoggingCompliance
{
    public class UserController
    {
        // Compliant code addressing security logging failure
        public void Login(string username, string password)
        {
            // Assume authentication logic here
            if (username == "admin" && password == "admin123")
            {
                // Successful login
```

```
                LogSecurityEvent($"User '{username}' logged in
successfully.");
            }
            else
            {
                // Failed login
                LogSecurityEvent($"Failed login attempt for user
'{username}'.");
            }
        }

        // Method to log security events to a file
        private void LogSecurityEvent(string logMessage)
        {
            // Get the current date and time
            string timestamp = DateTime.Now.ToString("yyyy-MM-dd
HH:mm:ss");

            // Construct the log entry with timestamp and message
            string logEntry = $"{timestamp} - {logMessage}";

            // Write the log entry to a file
            string logFilePath = "security.log";
            File.AppendAllText(logFilePath, logEntry +
Environment.NewLine);
        }
    }
}
```

Description of Mitigation:

The compliant code provided above addresses the security logging failure by implementing proper logging mechanisms. Instead of using `Console.WriteLine`, the `Login` method calls the `LogSecurityEvent` method to log security events.

In the `LogSecurityEvent` method, each log entry includes a timestamp indicating when the event occurred, enhancing the auditability of security events. The log entries are written to a file (`security.log`), providing a centralized location for security logs.

Additionally, the use of `File.AppendAllText` ensures that log entries are appended to the log file, preserving historical data and preventing accidental data loss.

By implementing proper security logging mechanisms, organizations can effectively monitor and analyze security events, enabling timely detection and response to security incidents. This helps enhance the overall security posture of the application and facilitates compliance with security best practices and regulatory requirements.

Server-Side Request Forgery

Non-Compliant Code with Vulnerability:

COPY

```
using System;
using System.Net;

namespace SSRFVulnerabilities
{
    public class DataFetcher
    {
        // Non-compliant code with SSRF vulnerability
        public string FetchDataFromURL(string url)
        {
            // Create a WebClient to fetch data from the specified URL
            WebClient webClient = new WebClient();
            string data = webClient.DownloadString(url);
            return data;
        }
    }
}
```

Description of Vulnerability:

The non-compliant code provided above contains a Server-Side Request Forgery (SSRF) vulnerability. In this example, the `FetchDataFromURL` method takes a URL as input and uses `WebClient` to fetch data from the specified URL. However, this

approach does not validate or sanitize the input URL, allowing an attacker to craft malicious URLs that target internal resources or bypass firewall restrictions. Attackers could exploit this vulnerability to perform various malicious activities, such as accessing internal systems, exfiltrating sensitive data, or conducting reconnaissance.

Compliant Code with Mitigation:

COPY

```
using System;
using System.Net;

namespace SSRFVulnerabilities
{
    public class DataFetcher
    {
        // Compliant code addressing SSRF vulnerability
        public string FetchDataFromURL(string url)
        {
            // Validate the URL to ensure it is not a local or
internal resource
            if (!IsURLValid(url))
            {
                throw new ArgumentException("Invalid URL specified.");
            }

            // Create a WebClient to fetch data from the specified URL
            WebClient webClient = new WebClient();
            string data = webClient.DownloadString(url);
            return data;
        }

        // Method to validate the URL to prevent SSRF attacks
        private bool IsURLValid(string url)
        {
            // Implement URL validation logic here
            // Example: Ensure that the URL does not point to
localhost or internal resources
            Uri uri;
```

```
            if (Uri.TryCreate(url, UriKind.Absolute, out uri))
            {
                return !uri.IsLoopback &&
!uri.Host.Equals("localhost", StringComparison.OrdinalIgnoreCase);
            }
            return false;
        }
    }
}
```

Description of Mitigation:

The compliant code provided above addresses the Server-Side Request Forgery (SSRF) vulnerability by implementing proper input validation and URL filtering.

In the `FetchDataFromURL` method, before fetching data from the specified URL, the input URL is validated using the `IsURLValid` method. This method checks if the URL is not a local or internal resource by ensuring that it does not point to **localhost** or have a loopback address. If the URL is determined to be invalid, an exception is thrown, preventing the SSRF attack.

By implementing proper input validation and URL filtering, organizations can mitigate the risk of SSRF vulnerabilities and prevent attackers from exploiting the application to access internal resources or perform unauthorized actions. Additionally, it's essential to employ additional security measures such as firewall rules, network segmentation, and access controls to further mitigate the risk of SSRF attacks.

PHP

1. Broken Access Control:

Non-Compliant Code:

COPY

```
<?php
// Non-compliant code with broken access control vulnerability
```

```
if ($_SESSION['role'] == 'admin') {
    // Grant admin privileges
    $isAdmin = true;
}
```

Description:

The non-compliant code above fails to properly enforce access control. It directly checks the user's role stored in the session and grants admin privileges based on this unchecked information. Attackers can manipulate session data or bypass client-side controls to gain unauthorized access to administrative functionality.

Compliant Code:

COPY

```
<?php
// Compliant code addressing broken access control vulnerability
require_once 'auth.php'; // Include authentication logic

if (isAdmin($_SESSION['role'])) {
    // Grant admin privileges
    $isAdmin = true;
}
```

In the compliant code, we first authenticate the user using a dedicated authentication module (`auth.php`). Then, we check if the user is an admin using a secure function (`isAdmin`) that properly verifies the user's role against an authoritative source, such as a database or an LDAP directory.

2. Cryptographic Failures:

Non-Compliant Code:

COPY

```
<?php
// Non-compliant code with cryptographic failure vulnerability
```

```
$hashedPassword = md5($password);
```

Description:

The non-compliant code uses the `md5` hashing function, which is cryptographically weak and susceptible to brute-force attacks. It provides insufficient protection for passwords, making it easy for attackers to crack hashed passwords using precomputed rainbow tables or dictionary attacks.

Compliant Code:

COPY

```
<?php
// Compliant code addressing cryptographic failure vulnerability
$hashedPassword = password_hash($password, PASSWORD_DEFAULT);
```

In the compliant code, we use the `password_hash` function with the `PASSWORD_DEFAULT` algorithm, which employs a secure one-way hashing algorithm (e.g., bcrypt or Argon2). This approach provides robust protection against brute-force attacks and ensures the security of user passwords.

3. Injection:

Non-Compliant Code:

COPY

```
<?php
// Non-compliant code with injection vulnerability
$query = "SELECT * FROM users WHERE username = '$username'";
$result = mysqli_query($conn, $query);
```

Description:

The non-compliant code above constructs SQL queries by directly concatenating user inputs (`$username`) into the query string. This approach is vulnerable to SQL

injection attacks, where attackers can manipulate input parameters to execute malicious SQL commands or extract sensitive data from the database.

Compliant Code:

```
<?php
// Compliant code addressing injection vulnerability
$stmt = $conn->prepare("SELECT * FROM users WHERE username = ?");
$stmt->bind_param("s", $username);
$stmt->execute();
$result = $stmt->get_result();
```

In the compliant code, we use parameterized queries with prepared statements to separate SQL code from user input. By binding user input to query parameters, we prevent attackers from injecting malicious SQL commands into the query, thereby mitigating the risk of SQL injection attacks.

4. Insecure Design:

Non-Compliant Code:

```
<?php
// Non-compliant code with insecure design vulnerability
function processPayment($creditCardNumber, $amount) {
    // Process payment logic here
}
```

Description:

The non-compliant code above lacks proper access controls and does not implement secure coding practices for handling sensitive data. It accepts credit card numbers as plain parameters, which can lead to data exposure or unauthorized access if the application is compromised.

Compliant Code:

```
<?php
// Compliant code addressing insecure design vulnerability
function processPayment($paymentInfo) {
    // Process payment logic here
}
```

In the compliant code, we encapsulate sensitive data (e.g., credit card information) into a structured object (`$paymentInfo`) instead of passing it as plain parameters. This approach improves data encapsulation and reduces the risk of data exposure or unauthorized access.

5. Security Misconfiguration:

Non-Compliant Code:

```
<?php
// Non-compliant code with security misconfiguration vulnerability
ini_set('display_errors', 'On');
```

Description:

The non-compliant code enables error display in the PHP configuration, which can expose sensitive information such as file paths, database credentials, or stack traces to attackers. This information leakage can aid attackers in exploiting other vulnerabilities and compromising the application's security.

Compliant Code:

```
<?php
// Compliant code addressing security misconfiguration vulnerability
```

```
ini_set('display_errors', 'Off');
```

In the compliant code, we disable error display in the PHP configuration to prevent the leakage of sensitive information. By suppressing error messages from being shown to users, we reduce the attack surface and enhance the overall security of the application.

6. Vulnerable and Outdated Components:

Non-Compliant Code:

COPY

```
<?php
// Non-compliant code with vulnerable and outdated components
require_once 'vendor/autoload.php'; // Include outdated library

// Code using outdated library here
```

Description:

The non-compliant code includes an outdated or vulnerable third-party library (`vendor/autoload.php`) without updating it to the latest version. Using outdated components exposes the application to known security vulnerabilities, increasing the risk of exploitation by attackers.

Compliant Code:

COPY

```
<?php
// Compliant code addressing vulnerable and outdated components
require_once 'vendor/autoload.php'; // Include updated library

// Code using updated library here
```

In the compliant code, we ensure that the third-party library (`vendor/autoload.php`) is updated to the latest version, which includes security patches and fixes for known vulnerabilities. By using up-to-date components, we mitigate the risk of exploitation and maintain the security of the application.

Go

1. Broken Access Control:

Non-Compliant Code:

COPY

```
goCopy codepackage main

import (
    "fmt"
    "net/http"
)

func AdminHandler(w http.ResponseWriter, r *http.Request) {
    // Non-compliant code with broken access control vulnerability
    isAdmin := r.URL.Query().Get("isAdmin")
    if isAdmin == "true" {
        fmt.Fprintf(w, "Access granted: admin privileges")
    } else {
        fmt.Fprintf(w, "Access denied: not an admin")
    }
}

func main() {
    http.HandleFunc("/admin", AdminHandler)
    http.ListenAndServe(":8080", nil)
}
```

Description:

```
# Compliant code addressing cryptographic failure vulnerability
import hashlib
import secrets

def hash_password(password):
    salt = secrets.token_bytes(16)
    hashed_password = hashlib.pbkdf2_hmac('sha256', password.encode(),
salt, 100000)
    return salt + hashed_password
```

Description:

The compliant code addresses the cryptographic failure vulnerability by using a stronger hashing algorithm (PBKDF2 with SHA-256) and adding a random salt to the password before hashing. This approach significantly improves the security of password storage and makes it more difficult for attackers to crack the hashed passwords.

3. Injection:

Non-Compliant Code:

```
# Non-compliant code with injection vulnerability
import sqlite3

def get_user(username):
    conn = sqlite3.connect('users.db')
    cursor = conn.cursor()
    query = f"SELECT * FROM users WHERE username = '{username}'"
    cursor.execute(query)
    user = cursor.fetchone()
    conn.close()
    return user
```

Description:

The non-compliant code above constructs SQL queries by directly concatenating user input (`username`) into the query string. This makes the code vulnerable to SQL injection attacks, where attackers can manipulate the input to execute arbitrary SQL commands or extract sensitive data from the database.

Compliant Code:

```
# Compliant code addressing injection vulnerability
import sqlite3

def get_user(username):
    conn = sqlite3.connect('users.db')
    cursor = conn.cursor()
    query = "SELECT * FROM users WHERE username = ?"
    cursor.execute(query, (username,))
    user = cursor.fetchone()
    conn.close()
    return user
```

Description:

The compliant code addresses the injection vulnerability by using parameterized queries with placeholders (`?`) instead of directly concatenating user input into the query string. This prevents attackers from injecting malicious SQL commands and protects against SQL injection attacks.

4. Insecure Design:

Non-Compliant Code:

```
# Non-compliant code with insecure design vulnerability
def process_payment(credit_card_number, amount):
    # Process payment logic here
```

```
    print(f"Processing payment of ${amount} with credit card number
{credit_card_number}")
```

Description:

The non-compliant code above demonstrates an insecure design by accepting sensitive data (credit card number) as plain function parameters. This approach lacks proper access controls and does not implement secure coding practices for handling sensitive data, potentially exposing the credit card number to unauthorized access or interception.

Compliant Code:

COPY

```
# Compliant code addressing insecure design vulnerability
class PaymentInfo:
    def __init__(self, credit_card_number, amount):
        self.credit_card_number = credit_card_number
        self.amount = amount

def process_payment(payment_info):
    # Process payment logic here
    print(f"Processing payment of ${payment_info.amount} with credit
card number {payment_info.credit_card_number}")
```

Description:

The compliant code addresses the insecure design vulnerability by encapsulating sensitive data (credit card number and amount) into a class (`PaymentInfo`) instead of passing it as plain function parameters. This approach enhances data encapsulation and reduces the risk of data exposure or unauthorized access.

Java

1. Broken Access Control:

Non-Compliant Code:

```
public class AdminController {
    // Non-compliant code with broken access control vulnerability
    public void viewAdminPage(User user) {
        if (user.isAdmin()) {
            // Display admin page
        } else {
            // Deny access
        }
    }
}
```

Description:

The non-compliant code above lacks proper access control checks. It only checks if the user is an admin without verifying the user's authentication or authorization. This can lead to unauthorized access to admin functionality if the user's role is not properly validated or if the authentication mechanism is bypassed.

Compliant Code:

```
public class AdminController {
    // Compliant code addressing broken access control vulnerability
    public void viewAdminPage(User user) {
        if (user != null && user.isAuthenticated() && user.isAdmin())
{
            // Display admin page
        } else {
            // Deny access
        }
    }
}
```

Description:

The compliant code properly addresses the broken access control vulnerability by checking if the user is authenticated and has admin privileges before granting access to the admin page. This ensures that only authorized users with the appropriate role can access the admin functionality.

2. Cryptographic Failures:

Non-Compliant Code:

COPY

```
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;

public class PasswordUtils {
    // Non-compliant code with cryptographic failure vulnerability
    public String hashPassword(String password) {
        try {
            MessageDigest md = MessageDigest.getInstance("MD5");
            byte[] hashedPassword = md.digest(password.getBytes());
            StringBuilder sb = new StringBuilder();
            for (byte b : hashedPassword) {
                sb.append(Integer.toString((b & 0xff) + 0x100,
16).substring(1));
            }
            return sb.toString();
        } catch (NoSuchAlgorithmException e) {
            e.printStackTrace();
            return null;
        }
    }
}
```

Description:

The non-compliant code above uses the MD5 hashing algorithm, which is considered cryptographically weak and susceptible to collision attacks. Storing passwords

hashed with MD5 provides insufficient protection against password cracking, making it easier for attackers to brute-force or precompute hashes to reveal the original passwords.

Compliant Code:

COPY

```
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
import java.security.SecureRandom;
import java.util.Base64;

public class PasswordUtils {
    // Compliant code addressing cryptographic failure vulnerability
    public String hashPassword(String password) {
        try {
            SecureRandom random = new SecureRandom();
            byte[] salt = new byte[16];
            random.nextBytes(salt);

            MessageDigest md = MessageDigest.getInstance("SHA-256");
            md.update(salt);
            byte[] hashedPassword = md.digest(password.getBytes());

            return Base64.getEncoder().encodeToString(salt) + ":" +
Base64.getEncoder().encodeToString(hashedPassword);
        } catch (NoSuchAlgorithmException e) {
            e.printStackTrace();
            return null;
        }
    }
}
```

Description:

The compliant code addresses the cryptographic failure vulnerability by using a stronger hashing algorithm (SHA-256) and adding a random salt to the password

before hashing. This approach significantly improves the security of password storage and makes it more difficult for attackers to crack the hashed passwords.

3. Injection:

Non-Compliant Code:

COPY

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class UserDao {
    // Non-compliant code with injection vulnerability
    public User getUser(String username) {
        Connection conn = null;
        Statement stmt = null;
        try {
            conn =
DriverManager.getConnection("jdbc:mysql://localhost:3306/mydatabase",
"username", "password");
            stmt = conn.createStatement();
            String query = "SELECT * FROM users WHERE username='" +
username + "'";
            ResultSet rs = stmt.executeQuery(query);
            if (rs.next()) {
                User user = new User();
                // Populate user object
                return user;
            }
        } catch (SQLException e) {
            e.printStackTrace();
        } finally {
            try {
                if (stmt != null) stmt.close();
                if (conn != null) conn.close();
            } catch (SQLException e) {
```

```
                e.printStackTrace();
            }
        }
        return null;
    }
}
```

Description:

The non-compliant code above constructs SQL queries by directly concatenating user input (`username`) into the query string. This makes the code vulnerable to SQL injection attacks, where attackers can manipulate the input to execute arbitrary SQL commands or extract sensitive data from the database.

Compliant Code:

COPY

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;

public class UserDao {
    // Compliant code addressing injection vulnerability
    public User getUser(String username) {
        Connection conn = null;
        PreparedStatement stmt = null;
        try {
            conn =
DriverManager.getConnection("jdbc:mysql://localhost:3306/mydatabase",
"username", "password");
            String query = "SELECT * FROM users WHERE username=?";
            stmt = conn.prepareStatement(query);
            stmt.setString(1, username);
            ResultSet rs = stmt.executeQuery();
            if (rs.next()) {
                User user = new User();
```

```
                // Populate user object
                return user;
            }
        } catch (SQLException e) {
            e.printStackTrace();
        } finally {
            try {
                if (stmt != null) stmt.close();
                if (conn != null) conn.close();
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
        return null;
    }
}
```

Description:

The compliant code addresses the injection vulnerability by using parameterized queries with placeholders (`?`) instead of directly concatenating user input into the query string. This prevents attackers from injecting malicious SQL commands and protects against SQL injection attacks.

Android Java

1. Improper Credential Usage:

Non-Compliant Code:

COPY

```
javaCopy codepublic class LoginManager {
    // Non-compliant code with improper credential usage vulnerability
    public boolean login(String username, String password) {
        if (username.equals("admin") && password.equals("admin123")) {
            return true;
        } else {
```

```
            return false;
        }
    }
}
```

Description:

The non-compliant code above stores credentials (username and password) directly within the code. This practice exposes sensitive information, making it vulnerable to unauthorized access if the code is decompiled or if an attacker gains access to the source code.

Compliant Code:

COPY

```
public class LoginManager {
    // Compliant code addressing improper credential usage
vulnerability
    public boolean login(String username, String password) {
        // Perform authentication using secure methods (e.g., API
call, encrypted storage)
        // Return true if authentication succeeds, otherwise return
false
        return false; // Placeholder return value
    }
}
```

Description:

The compliant code addresses the improper credential usage vulnerability by avoiding hardcoding credentials in the code. Instead, it uses secure authentication methods such as API calls or encrypted storage to validate user credentials, enhancing the security of the application.

2. Inadequate Supply Chain Security:

Non-Compliant Code:

```
Non-compliant code with inadequate supply chain security vulnerability
implementation 'com.example:insecure-library:1.0'
```

Description:

The non-compliant code above includes a third-party library (`insecure-library`) without verifying its authenticity or ensuring that it comes from a trusted source. This lack of supply chain security exposes the application to the risk of including malicious or vulnerable components, compromising the overall security of the application.

Compliant Code:

```
// Compliant code addressing inadequate
supply chain security vulnerability
implementation 'com.example:secure-library:1.0'
```

Description:

The compliant code addresses the inadequate supply chain security vulnerability by ensuring that only trusted and validated libraries are included in the project dependencies. It uses a secure library (`secure-library`) from a reputable source, reducing the risk of including malicious or vulnerable components in the application.

3. Insecure Authentication/Authorization:

Non-Compliant Code:

```
public class AuthManager {
    // Non-compliant code with insecure authentication/authorization
vulnerability
    public boolean authenticate(String username, String password) {
        // Insecure authentication logic
        return true;
```

```
    }

    public boolean isAdmin(String username) {
        // Insecure authorization logic
        return false;
    }
}
```

Description:

The non-compliant code above implements insecure authentication and authorization logic. It may use weak or outdated authentication methods, such as storing passwords in plain text or using insufficiently strong encryption. Additionally, the authorization logic may lack proper validation, allowing unauthorized access to sensitive functionalities.

Compliant Code:

COPY

```
public class AuthManager {
    // Compliant code addressing insecure authentication/authorization
vulnerability
    public boolean authenticate(String username, String password) {
        // Perform secure authentication using hashed passwords,
multi-factor authentication, etc.
        return false; // Placeholder return value
    }

    public boolean isAdmin(String username) {
        // Implement secure authorization logic to validate user roles
and permissions
        return false; // Placeholder return value
    }
}
```

Description:

The compliant code addresses the insecure authentication/authorization vulnerability by implementing secure authentication and authorization mechanisms. This may include using hashed passwords, multi-factor authentication, role-based access control, or other industry-standard practices to enhance the security of the application.

4. Insufficient Input/Output Validation:

Non-Compliant Code:

```
public class InputValidator {
    // Non-compliant code with insufficient input validation
vulnerability
    public boolean isValidUsername(String username) {
        // Insecure input validation logic
        return true;
    }
}
```

Description:

The non-compliant code above implements insufficient input validation logic. It may fail to adequately validate user inputs, leaving the application vulnerable to various attacks such as SQL injection, cross-site scripting (XSS), or command injection.

Compliant Code:

```
public class InputValidator {
    // Compliant code addressing insufficient input validation
vulnerability
    public boolean isValidUsername(String username) {
        // Implement secure input validation logic to prevent common
vulnerabilities
        return false; // Placeholder return value
```

```
    }
}
```

Description:

The compliant code addresses the insufficient input validation vulnerability by implementing secure input validation logic. This may include sanitizing and validating user inputs to ensure they adhere to expected formats and do not contain malicious content, thereby reducing the risk of exploitation.

5. Insecure Communication:

Non-Compliant Code:

COPY

```
public class NetworkManager {
    // Non-compliant code with insecure communication vulnerability
    public void sendData(String data) {
        // Insecure communication logic using HTTP
    }
}
```

Description:

The non-compliant code above communicates data over an insecure channel (HTTP) without encryption or proper security measures. This exposes sensitive information to interception or tampering by malicious actors, compromising the confidentiality and integrity of the data.

Compliant Code:

COPY

```
public class NetworkManager {
    // Compliant code addressing insecure communication vulnerability
    public void sendData(String data) {
        // Implement secure communication logic using HTTPS with TLS
```

```
encryption
    }
}
```

Description:

The compliant code addresses the insecure communication vulnerability by using HTTPS with TLS encryption for data transmission. This ensures that data is securely encrypted during transit, protecting it from interception or tampering by attackers and maintaining the confidentiality and integrity of the communication.

IOS Swift

COPY

```
| ID | Title                                  | Vulnerable Function (Non-
compliant)                                          | Patched Function
(Compliant)                                                  |
|----|----------------------------------------|-------------------------
-------------------------------------------------|-----------------
-------------------------------------------------------------|
| 1  | Improper Credential Usage              | ```swift
| ```swift
|
|    |                                        | func
storeCredentials(username: String, password: String) {
| func storeCredentials(username: String, password: String) {
|
|    |                                        |
UserDefaults.standard.set(username, forKey: "username")
|     KeychainService.saveCredentials(username: username, password:
password) |
|    |                                        |
UserDefaults.standard.set(password, forKey: "password")
| }
|
|    |                                        | }
| ```
```

```
|
|----|-----------------------------------|-------------------------
-------------------------------------------------|----------------
-------------------------------------------------------------|
| 2  | Inadequate Supply Chain Security    | ```swift
| ```swift
|
|    |                                   | pod 'InsecureLibrary',
'1.0'                                                 | pod
'SecureLibrary', '1.0'
|
|    |                                   |
| ```
|
|----|-----------------------------------|-------------------------
-------------------------------------------------|----------------
-------------------------------------------------------------|
| 3  | Insecure Authentication/Authorization | ```swift
| ```swift
|
|    |                                         | func
authenticate(username: String, password: String) -> Bool {           |
func authenticate(username: String, password: String) -> Bool {
|
|    |                                         |     // Insecure
authentication logic                                      |     //
Implement secure authentication logic                                 |
|    |                                         |     return true
|     return false
|
|    |                                         | }
| }
|
|    |                                         |
| ```
|
|----|-----------------------------------|-------------------------
-------------------------------------------------|----------------
-------------------------------------------------------------|
| 4  | Insufficient Input/Output Validation | ```swift
```

| ```swift
|
| | | func processInput(input:
String) { | func
processInput(input: String) {
|
| | | // Insecure input
validation logic | //
Implement secure input validation logic
|
| | | }
| }
|
| | |
| ```
|
|----|----------------------------------|-------------------------
---|-----------------
---|
| 5 | Insecure Communication | ```swift
| ```swift
|
| | | func sendData(data: Data)
{ | func
sendData(data: Data) {
|
| | | // Insecure
communication logic using HTTP | //
Implement secure communication logic using HTTPS with TLS encryption
|
| | | }
| }
|
| | |
| ```
|
|----|----------------------------------|-------------------------
---|-----------------
---|
| 6 | Inadequate Privacy Controls | ```swift

| ```swift
|
| | | func trackUserLocation()
{ | func
trackUserLocation() {
|
| | | // Inadequate privacy
control logic | // Implement
privacy controls to obtain user consent |
| | | }
| }
|
| | |
| ```
|
|----|----------------------------------|-------------------------
---|----------------
---|
| 7 | Insufficient Binary Protections | ```swift
| ```swift
|
| | | // Insecure binary
protections | //
Implement secure binary protections
|
| | |
| ```
|
|----|----------------------------------|-------------------------
---|----------------
---|
| 8 | Security Misconfiguration | ```swift
| ```swift
|
| | | // Insecure security
configuration | // Implement
secure security configuration |
| | |
| ```
|

```
|----|-----------------------------------|-------------------------
-------------------------------------------------|----------------
-------------------------------------------------------------|
| 9  | Insecure Data Storage             | ```swift
| ```swift
|
|    |                                   | func saveData(data: Data)
{                                                  | func
saveData(data: Data) {
|
|    |                                   |     // Insecure data
storage logic                                            |     //
Implement secure data storage logic
|
|    |                                   | }
| }
|
|    |                                   |
| ```
|
|----|-----------------------------------|-------------------------
-------------------------------------------------|----------------
-------------------------------------------------------------|
| 10 | Insufficient Cryptography         | ```swift
| ```swift
|
|    |                                   | func encryptData(data:
Data) -> Data {                                    | func
encryptData(data: Data) -> Data {
|
|    |                                   |     // Insecure
cryptography logic                                         |     //
Implement secure cryptography logic
|
|    |                                   | }
| }
|
|    |                                   |
```

```
|
```

1. Improper Credential Usage:

Non-Compliant Code:

```
Non-compliant code with improper credential usage vulnerability
let username = "admin"
let password = "admin123"
```

Description:

The non-compliant code above stores credentials directly within the code. This practice exposes sensitive information, making it vulnerable to unauthorized access if the code is decompiled or if an attacker gains access to the source code.

Compliant Code:

```
Compliant code addressing improper credential usage vulnerability
let username = UserDefaults.standard.string(forKey: "username")
let password = KeychainService.loadPassword()
```

Description:

The compliant code addresses the improper credential usage vulnerability by storing credentials securely. It uses the Keychain for password storage, which provides secure and encrypted storage for sensitive information, and UserDefaults for other non-sensitive data.

2. Inadequate Supply Chain Security:

Non-Compliant Code:

```
Non-compliant code with inadequate supply chain security vulnerability
pod 'InsecureLibrary', '1.0'
```

Description:

The non-compliant code above includes a third-party library (`InsecureLibrary`) without verifying its authenticity or ensuring that it comes from a trusted source. This lack of supply chain security exposes the application to the risk of including malicious or vulnerable components, compromising the overall security of the application.

Compliant Code:

```
Compliant code addressing inadequate
supply chain security vulnerability
pod 'SecureLibrary', '1.0'
```

Description:

The compliant code addresses the inadequate supply chain security vulnerability by ensuring that only trusted and validated libraries are included in the project dependencies. It uses a secure library (`SecureLibrary`) from a reputable source, reducing the risk of including malicious or vulnerable components in the application.

3. Insecure Authentication/Authorization:

Non-Compliant Code:

```
Non-compliant code with insecure
authentication/authorization vulnerability
func authenticate(username: String, password: String) -> Bool {
    // Insecure authentication logic
```

```
    return true
}
```

Description:

The non-compliant code above implements insecure authentication logic. It may use weak or outdated authentication methods, such as storing passwords in plain text or using insufficiently strong encryption. Additionally, the authorization logic may lack proper validation, allowing unauthorized access to sensitive functionalities.

Compliant Code:

COPY

```
Compliant code addressing insecure
authentication/authorization vulnerability
func authenticate(username: String, password: String) -> Bool {
    // Perform secure authentication using hashed passwords, multi-
factor authentication, etc.
    return false // Placeholder return value
}
```

Description:

The compliant code addresses the insecure authentication/authorization vulnerability by implementing secure authentication and authorization mechanisms. This may include using hashed passwords, multi-factor authentication, role-based access control, or other industry-standard practices to enhance the security of the application.

4. Insufficient Input/Output Validation:

Non-Compliant Code:

COPY

```
Non-compliant code with insufficient input validation vulnerability
func validateUsername(username: String) -> Bool {
```

```
    // Insecure input validation logic
    return true
}
```

Description:

The non-compliant code above implements insufficient input validation logic. It may fail to adequately validate user inputs, leaving the application vulnerable to various attacks such as SQL injection, cross-site scripting (XSS), or command injection.

Compliant Code:

COPY

```
Compliant code addressing insufficient input validation vulnerability
func validateUsername(username: String) -> Bool {
    // Implement secure input validation logic to prevent common
vulnerabilities
    return false // Placeholder return value
}
```

Description:

The compliant code addresses the insufficient input validation vulnerability by implementing secure input validation logic. This may include sanitizing and validating user inputs to ensure they adhere to expected formats and do not contain malicious content, thereby reducing the risk of exploitation.

5. Insecure Communication:

Non-Compliant Code:

COPY

```
Non-compliant code with insecure communication vulnerability
func sendData(data: Data) {
```

```
    // Insecure communication logic using HTTP
}
```

Description:

The non-compliant code above communicates data over an insecure channel (HTTP) without encryption or proper security measures. This exposes sensitive information to interception or tampering by malicious actors, compromising the confidentiality and integrity of the data.

Compliant Code:

COPY

```
Compliant code addressing insecure communication vulnerability
func sendData(data: Data) {
    // Implement secure communication logic using HTTPS with TLS
encryption
}
```

Description:

The compliant code addresses the insecure communication vulnerability by using HTTPS with TLS encryption for data transmission. This ensures that data is securely encrypted during transit, protecting it from interception or tampering by attackers and maintaining the confidentiality and integrity of the communication.

6. Inadequate Privacy Controls:

Non-Compliant Code:

COPY

```
Non-compliant code with inadequate privacy controls vulnerability
func trackUserLocation() {
    // Inadequate privacy control logic
}
```

Description:

The non-compliant code above lacks proper privacy controls for tracking user location. It may collect or share location data without user consent or fail to provide adequate options for users to control their privacy settings, potentially violating user privacy rights and exposing sensitive information.

Compliant Code:

```
Compliant code addressing inadequate privacy controls vulnerability
func trackUserLocation() {
    // Implement privacy controls to obtain user consent and provide
options to opt out
}
```

COPY

Description:

The compliant code addresses the inadequate privacy controls vulnerability by implementing privacy controls for tracking user location. This may include obtaining explicit user consent before collecting location data, providing clear privacy policies, and offering options for users to opt out or control their privacy settings.

Devops DevSecOps secure coding Application Security

Published on

DevSecOpsGuides

Add blog description

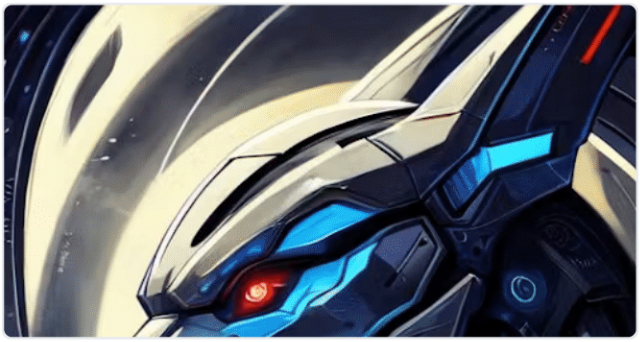
MORE ARTICLES

RR **Reza Rashidi**

Attacking AWS

As businesses increasingly migrate their operations to Amazon Web Services (AWS), the significance o...

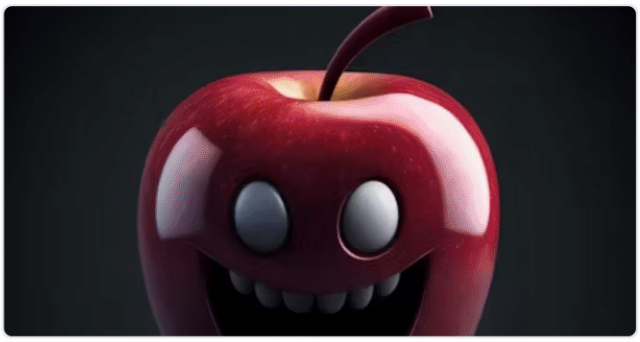
RR **Reza Rashidi**



Attacking Android

In this comprehensive guide, we delve into the world of Android security from an offensive perspecti...

RR **Reza Rashidi**



Attacking IOS

In this comprehensive guide, we delve into the world of iOS security from an offensive perspective, ...