ATTACKING APIS

WITH SECURE CODING PRACTICES IN .NET, JAVA

WWW.DEVSECOPSGUIDES.COM

DevSecOpsGuides

Follow

Attacking APIs

Feb 19, 2024 · 34 min read

Table of contents

fuzzing on API endpoints

Attacking JWTs: Techniques, Tools, and Countermeasures

- Understanding JWTs:
- Common Attacks on JWTs:
- Defending Against JWT Attacks:

- Understanding JWTs:
- Common Attacks on JWTs:
- Defending Against JWT Attacks:

Non-compliant Code Examples

- Java

- Java

Compliant Code Examples

- Java

- Java

Bypassing Authentication Controls

Non-compliant Code Examples

- Java
- .NET (C#)

- Java
- .NET (C#)

Compliant Code Examples

- .NET (C#)
- Java

- .NET (C#)
- Java

Credential Attacks: Strategies and Techniques

- Credential Stuffing
- Password Spraying
- Brute Forcing

- Credential Stuffing
- Password Spraying
- Brute Forcing

Key-Based Attacks: Exploiting Vulnerabilities in API Keys

- Hardcoded Keys
- Leaked Keys
- Weak Keys

- Hardcoded Keys
- Leaked Keys
- Weak Keys

Non-compliant Code Examples

- .NET (C#)
- Java

- .NET (C#)
- Java

Compliant Code Examples

- .NET (C#)
- Java

- Attack Scenario:
- Mitigation Techniques:
- Non-compliant Code Examples
  - .NET (C#)
  - Java
- Compliant Code Examples
  - .NET (C#)
  - Java
- Object-Level Authorization (BOLA)
  - Mitigation:
- Function-Level Authorization (BFLA)
  - Mitigation:
- Non-Compliant Code Examples for BOLA (Broken Object-Level Authorization)
  - .NET (C#)
  - Java
- Compliant Code Examples for BOLA
  - .NET (C#)
  - Java
- Data Attacks
  - Excessive Information Exposure:
  - Mass Assignment:
  - Mitigation:
- Non-Compliant Code Examples for Excessive Information Exposure
  - .NET (C#)

Show less

APIs (Application Programming Interfaces) have become integral components of modern software systems, facilitating communication and interaction between various applications and services. However, they also represent a significant attack surface, susceptible to a variety of malicious activities. This article explores common attack vectors targeting APIs, along with secure coding examples to mitigate these vulnerabilities.

Finding API Keys:

1. **Using an Intercepting Proxy:**
   - Tools like Burp Suite or OWASP ZAP can be used as intercepting proxies.
   - Configure the proxy to intercept network traffic.
   - Install a trusted certificate on the client device if TLS (HTTPS) is used.

2. **Discovering Keys and Tokens:**
    - API keys or tokens can be found in various locations:
        - Open S3 buckets on AWS
        - Public GitHub repositories and gists
        - Pastebin pastes (https://pastebin.com/)
        - SharePoint, Confluence, and Jira sites
        - Log files
        - CI/CD systems (as unprotected variables)
        - API request URLs
3. **Verifying Key Validity:**
    - After discovering a key or token, verify its validity by querying the relevant service endpoint.
    - For example, to test a Facebook token:

```
curl -X GET
"https://developers.facebook.com/tools/debug/accesstoken/?
access_token=ACCESS_TOKEN_HERE&version=v3.2"
```

1. **Using TruffleHog for GitHub Repositories:**
    - TruffleHog is a tool for retrieving keys and tokens from GitHub repositories.
    - Run TruffleHog in a Docker container:

```
docker run --platform linux/arm64 -it -v "$PWD:/pwd"
trufflesecurity/trufflehog:latest github --
repo https://github.com/trufflesecurity/test_keys
```

Enumeration and Discovery of APIs:

1. **Using Kiterunner:**
    - Kiterunner is a tool for discovering API endpoints.
    - It uses proprietary databases built from OpenAPI definitions.
    - Run Kiterunner against a target endpoint using a Kite file:

COPY

```
kiterunner scan -f kite_file.yaml -
u http://api.example.com
```

1. **Kiterunner Output Example:**
    - After scanning, Kiterunner provides details about detected endpoints, including their existence and status codes.

COPY

```
POST 202 [ 22, 4, 1] http://api.example.com/api/register
0cc39f7908a098262e1a94331f0711f969c08024

POST 401 [ 38, 4, 1] http://api.example.com/api/login
0cc39f753802733afd682ea9673e6861e685415
```

1. **Additional Options:**
    - Kiterunner can operate in scan mode using OpenAPI definitions or in brute mode using word lists.
    - It can be constrained to enumerate endpoints to a specified depth to improve scan times.

fuzzing on API endpoints

To perform fuzzing on API endpoints using the wfuzz tool, follow these steps:

1. **Prepare Word List:**
    - Create a word list containing potential inputs to be fuzzed against the API endpoints. You can obtain word lists from various sources, such as SecLists by Daniel Miessler.
2. **Run wfuzz in Docker Container:**
    - Execute the following command to run wfuzz in a Docker container, specifying options for fuzzing:

COPY

```
docker run --rm -v $(pwd)/wordlist:/wordlist/ -it
ghcr.io/xmendez/wfuzz wfuzz -w wordlist/general/common.txt
--hc 404 http://192.168.16.23:8090/FUZZ
```

- Explanation of options:
    - `--rm`: Remove the container upon termination.

- `-v $(pwd)/wordlist:/wordlist/` : Mount the local word list directory into the container.
- `-it` : Use an interactive terminal session.
- `ghcr.io/xmendez/wfuzz` : Name of the Docker container repository and image.
- `-w wordlist/general/common.txt` : Specify the word list file to be used.
- `--hc 404` : Suppress responses with a 404 status code.
- `http://192.168.16.23:8090/FUZZ` : Specify the target host, port, and path for fuzzing. The `FUZZ` keyword will be replaced with each word from the word list.

1. **Interpret Results:**
    - Analyze the output generated by wfuzz to identify any unexpected responses, which may indicate vulnerabilities in the API endpoints.

Regarding restler-fuzzer, it is primarily designed to scan APIs against OpenAPI definitions to identify vulnerabilities. Here's a brief overview of using restler-fuzzer:

1. **Prepare OpenAPI Definition:**
    - Have an OpenAPI (formerly known as Swagger) definition of the API you want to test.
2. **Run restler-fuzzer:**
    - Execute restler-fuzzer, providing the path to the OpenAPI definition as input.

3. **Interpret Results:**
    - Analyze the output to identify any vulnerabilities detected by restler-fuzzer.

Attacking JWTs: Techniques, Tools, and Countermeasures

JSON Web Tokens (JWTs) have become a popular method for authentication and authorization in modern web applications. However, like any authentication mechanism, JWTs are susceptible to various attacks. In this article, we'll explore common attacks targeting JWTs, along with tools and techniques used by attackers, as well as countermeasures to mitigate these threats.

Understanding JWTs:

JWTs consist of three parts: a header, a payload, and a signature. The header typically contains metadata about the token, the payload contains claims, and the signature is used to verify the authenticity of the token.

Common Attacks on JWTs:

1. **JWT Verification Bypass:**
    - Attackers attempt to bypass JWT verification by tampering with the token's header or signature.
    - *Example Command:* Using jwt_tool to analyze and tamper with JWTs.

      ```
      jwt_tool <JWT>
      ```

2. **HMAC Key Cracking:**

- Attackers try to crack the HMAC secret key used to sign JWTs, allowing them to generate valid tokens.
- *Example Command:* Using jwt_tool to crack the HMAC secret key.

  ```
  jwt_tool --crack-hmac <JWT>
  ```

3. **Key Confusion Attacks:**
   - Attackers exploit weaknesses in asymmetric ciphers by confusing the application into using a different public key.
   - *Example Command:* Using jwt_tool to perform key confusion attacks.

     ```
     jwt_tool --key-confusion <JWT>
     ```

4. **None Algorithm Exploitation:**
   - Attackers force the use of the "none" algorithm to create unsigned and unvalidated JWTs.
   - *Example Command:* Using jwt_tool to spoof JWTs with the "none" algorithm.

     ```
     jwt_tool --none <JWT>
     ```

5. **Spoofing JWKS:**
   - Attackers spoof JSON Web Key Sets (JWKS) to impersonate legitimate keys and manipulate token validation.
   - *Example Command:* Using jwt_tool to spoof JWKS.

     ```
     jwt_tool --spoof-jwks <JWT>
     ```

Defending Against JWT Attacks:

1. **Use Strong Algorithms:**

- Employ strong cryptographic algorithms (e.g., RSA, ECDSA) instead of weak ones like HMAC.

2. **Validate JWTs Properly:**
    - Implement strict validation of JWTs, including checking the algorithm used, verifying the signature, and validating claims.
3. **Rotate Keys Regularly:**
    - Rotate HMAC secret keys and RSA key pairs regularly to minimize the impact of key compromise.
4. **Enforce Token Expiry:**
    - Set expiration times for JWTs and enforce token expiration policies to mitigate the risk of token reuse.
5. **Implement Rate Limiting:**
    - Enforce rate limiting on JWT-related endpoints to prevent brute-force and denial-of-service attacks.

Non-compliant Code Examples

COPY

```
using System;
using System.IdentityModel.Tokens.Jwt;
using System.Text;
using Microsoft.IdentityModel.Tokens;

public class JWTVerifier
{
    public static bool VerifyToken(string token)
    {
```

```
        try
        {
            var tokenHandler = new
JwtSecurityTokenHandler();
            var validationParameters = new
TokenValidationParameters
            {
                ValidateIssuerSigningKey = false, //
Vulnerable: Skipping key validation
                ValidateIssuer = false, // Vulnerable:
Skipping issuer validation
                ValidateAudience = false // Vulnerable:
Skipping audience validation
            };

            SecurityToken validatedToken;
            tokenHandler.ValidateToken(token,
validationParameters, out validatedToken);
            return true;
        }
        catch (Exception)
        {
            return false;
        }
    }
}
```

Java

COPY

```
import io.jsonwebtoken.Claims;
import io.jsonwebtoken.JwtException;
import io.jsonwebtoken.Jwts;
```

```
public class JWTVerifier {
    public static boolean verifyToken(String token) {
        try {
            Claims claims =
Jwts.parser().parseClaimsJwt(token).getBody();
            return true;
        } catch (JwtException e) {
            return false;
        }
    }
}
```

Compliant Code Examples

COPY

```
using System;
using System.IdentityModel.Tokens.Jwt;
using System.Text;
using Microsoft.IdentityModel.Tokens;

public class JWTVerifier
{
    private static readonly string SecretKey = "your-
secure-hmac-secret-key";
    private static readonly string Issuer = "your-issuer";
    private static readonly string Audience = "your-
audience";

    public static bool VerifyToken(string token)
    {
        try
```

```
        {
            var tokenHandler = new
JwtSecurityTokenHandler();
            var validationParameters = new
TokenValidationParameters
            {
                ValidateIssuerSigningKey = true,
                IssuerSigningKey = new
SymmetricSecurityKey(Encoding.UTF8.GetBytes(SecretKey)),
                ValidateIssuer = true,
                ValidIssuer = Issuer,
                ValidateAudience = true,
                ValidAudience = Audience,
                ValidateLifetime = true,
                ClockSkew = TimeSpan.Zero
            };

            SecurityToken validatedToken;
            tokenHandler.ValidateToken(token,
validationParameters, out validatedToken);
            return true;
        }
        catch (Exception)
        {
            return false;
        }
    }
}
```

Java

COPY

```
import io.jsonwebtoken.Claims;
import io.jsonwebtoken.JwtException;
import io.jsonwebtoken.Jwts;
import io.jsonwebtoken.security.Keys;

import javax.crypto.SecretKey;
import java.util.Date;

public class JWTVerifier {
    private static final String SECRET_KEY = "your-secure-
hmac-secret-key";
    private static final String ISSUER = "your-issuer";
    private static final String AUDIENCE = "your-
audience";

    public static boolean verifyToken(String token) {
        try {
            SecretKey key =
Keys.hmacShaKeyFor(SECRET_KEY.getBytes());
            Claims claims = Jwts.parserBuilder()
                    .setSigningKey(key)
                    .requireIssuer(ISSUER)
                    .requireAudience(AUDIENCE)
                    .build()
                    .parseClaimsJws(token)
                    .getBody();
            return true;
        } catch (JwtException e) {
            return false;
        }
    }
}
```

The non-compliant code examples demonstrate insecure JWT verification practices, such as skipping key, issuer, and audience validation. These vulnerabilities can lead to various attacks, including JWT verification bypass, HMAC key cracking, and key confusion attacks.

In the compliant code examples:

- The .NET code securely validates the JWT by setting parameters to validate the issuer, audience, and the cryptographic signature of the token. It uses a secure HMAC secret key and specifies the issuer and audience to ensure that only tokens from trusted sources are accepted.
- The Java code uses the `Jwts.parserBuilder()` method to construct a parser with explicit settings for the secret key, issuer, and audience. It requires the token to be signed with the specified HMAC secret key and validates the issuer and audience to prevent unauthorized tokens from being accepted.

These compliant implementations enhance security by properly validating JWTs, mitigating the risk of various attacks outlined in the original prompt.

Bypassing Authentication Controls

One of the primary goals of attackers targeting APIs is to bypass authentication controls to gain unauthorized access. This can be achieved through several methods, including:

- **Guessing Access Credentials:** Attackers may attempt to guess usernames and passwords to gain access to API endpoints.

- **Stealing or Forging Credentials:** Illegitimate acquisition or fabrication of credentials to impersonate legitimate users.
- **Exploiting Weak Authentication Logic:** Identifying vulnerabilities or loopholes in the authentication logic of APIs to circumvent security measures.

Non-compliant Code Examples

COPY

```
using System;
using System.Net;
using System.Net.Http;
using System.Threading.Tasks;

public class APIClient
{
    private readonly HttpClient _httpClient;

    public APIClient()
    {
        _httpClient = new HttpClient();
    }

    public async Task<string> Login(string username,
string password)
    {
        var request = new
HttpRequestMessage(HttpMethod.Post,
"https://api.example.com/login");
        request.Content = new FormUrlEncodedContent(new[]
        {
            new KeyValuePair<string, string>("username",
```

```
username),
            new KeyValuePair<string, string>("password",
password)
        });

        var response = await
_httpClient.SendAsync(request);

        if (response.IsSuccessStatusCode)
        {
            return await
response.Content.ReadAsStringAsync();
        }
        else
        {
            return null;
        }
    }
}
```

Java

.NET (C#)

COPY

```
import java.io.IOException;
import java.net.HttpURLConnection;
import java.net.URL;
import java.util.HashMap;
import java.util.Map;

public class APIClient {
    private final String BASE_URL =
"https://api.example.com";
```

```
    public String login(String username, String password)
throws IOException {
        URL url = new URL(BASE_URL + "/login");
        HttpURLConnection con = (HttpURLConnection)
url.openConnection();
        con.setRequestMethod("POST");
        con.setDoOutput(true);

        Map<String, String> parameters = new HashMap<>();
        parameters.put("username", username);
        parameters.put("password", password);

        StringBuilder postData = new StringBuilder();
        for (Map.Entry<String, String> param :
parameters.entrySet()) {
            if (postData.length() != 0)
postData.append('&');
            postData.append(param.getKey());
            postData.append('=');
            postData.append(param.getValue());
        }
        byte[] postDataBytes =
postData.toString().getBytes("UTF-8");
        con.getOutputStream().write(postDataBytes);

        int responseCode = con.getResponseCode();
        if (responseCode == HttpURLConnection.HTTP_OK) {
            // Read and return response
        } else {
            // Handle error
            return null;
        }
```

```
    }
}
```

Compliant Code Examples

.NET (C#)

COPY

```
using System;
using System.Net;
using System.Net.Http;
using System.Threading.Tasks;

public class APIClient
{
    private readonly HttpClient _httpClient;

    public APIClient()
    {
        _httpClient = new HttpClient();
    }

    public async Task<string> Login(string username,
string password)
    {
        var credentials = new FormUrlEncodedContent(new[]
        {
            new KeyValuePair<string, string>("username",
username),
            new KeyValuePair<string, string>("password",
password)
        });
```

```
        var response = await
_httpClient.PostAsync("https://api.example.com/login",
credentials);

        if (response.IsSuccessStatusCode)
        {
            return await
response.Content.ReadAsStringAsync();
        }
        else
        {
            return null;
        }
    }
}
```

Java

COPY

```
import java.io.IOException;
import java.net.HttpURLConnection;
import java.net.URL;
import java.util.HashMap;
import java.util.Map;

public class APIClient {
    private final String BASE_URL =
"https://api.example.com";

    public String login(String username, String password)
throws IOException {
        URL url = new URL(BASE_URL + "/login");
        HttpURLConnection con = (HttpURLConnection)
```

```
url.openConnection();
        con.setRequestMethod("POST");
        con.setDoOutput(true);

        Map<String, String> parameters = new HashMap<>();
        parameters.put("username", username);
        parameters.put("password", password);

        StringBuilder postData = new StringBuilder();
        for (Map.Entry<String, String> param :
parameters.entrySet()) {
            if (postData.length() != 0)
postData.append('&');
            postData.append(param.getKey());
            postData.append('=');
            postData.append(param.getValue());
        }
        byte[] postDataBytes =
postData.toString().getBytes("UTF-8");
        con.getOutputStream().write(postDataBytes);

        int responseCode = con.getResponseCode();
        if (responseCode == HttpURLConnection.HTTP_OK) {
            // Read and return response
        } else {
            // Handle error
            return null;
        }
    }
}
```

The non-compliant code examples for both .NET and Java demonstrate insecure practices in API authentication, specifically in the

`Login` method. These examples send sensitive data (username and password) as plain text in the request body, which is vulnerable to interception and exploitation by attackers.

In the compliant code examples:

- Both the .NET and Java examples have been updated to use HTTPS for secure communication with the API endpoint.
- The .NET example uses `HttpClient` to send a POST request with the credentials sent in the request body using `FormUrlEncodedContent`.
- The Java example constructs the request body as a URL-encoded string and sends it in the request body using `HttpURLConnection`.

By utilizing HTTPS and sending credentials securely in the request body, the compliant code examples improve the security of the authentication process, mitigating the risk of unauthorized access through credential guessing, stealing, or forging, as well as exploitation of weak authentication logic.

Credential Attacks: Strategies and Techniques

Credential Stuffing

Credential stuffing involves leveraging compromised credentials obtained from data breaches to gain unauthorized access to systems. Attackers exploit the human tendency to reuse passwords across multiple platforms.

Mitigation: Implement multi-factor authentication (MFA), monitor compromised email addresses, and employ bot detection mechanisms.

Password Spraying

Password spraying attacks utilize common sets of usernames and passwords to launch attacks against login endpoints. Unlike brute force attacks, password spraying minimizes the risk of account lockouts by making few attempts per user.

Mitigation: Implement strong password policies, enforce account lockout mechanisms, and monitor login attempts.

Brute Forcing

Brute force attacks involve systematically trying every possible combination of usernames and passwords to gain access. Attackers utilize large wordlists and automated tools to maximize their chances of success.

Mitigation: Implement account lockout policies, utilize CAPTCHA mechanisms, and enforce strong password complexity requirements.

Key-Based Attacks: Exploiting Vulnerabilities in API Keys

API keys serve as credentials to authenticate clients accessing APIs. Attackers exploit weaknesses in key management and implementation to compromise API security.

Hardcoded Keys

Hardcoded keys embedded in application configurations or binaries present a significant security risk. Attackers can easily retrieve hardcoded keys from application binaries, leading to unauthorized access.

Mitigation: Utilize secure key management practices, avoid hardcoding keys in application code, and employ encryption for sensitive data.

Leaked Keys

Leaked keys, often due to insecure source code repositories or documentation systems, provide attackers with access to API endpoints. Keys may also be leaked by third-party platforms or disgruntled insiders.

Mitigation: Secure access to source code repositories, regularly audit access controls, and monitor for unauthorized key usage.

Weak Keys

Weak keys, characterized by inadequate randomness or predictability, are susceptible to brute force or cracking attacks. Attackers exploit weaknesses in key generation algorithms to compromise API security.

Mitigation: Utilize strong cryptographic algorithms, enforce key length and complexity requirements, and regularly rotate keys.

Non-compliant Code Examples

.NET (C#)

COPY

```
using System;
using System.Net;
using System.Net.Http;
using System.Threading.Tasks;

public class APIClient
{
```

```
    private readonly HttpClient _httpClient;
    private readonly string _apiKey = "YOUR_API_KEY";

    public APIClient()
    {
        _httpClient = new HttpClient();
    }

    public async Task<string> GetData(string endpoint)
    {
        var request = new
HttpRequestMessage(HttpMethod.Get,
$"https://api.example.com/{endpoint}");
        request.Headers.Add("x-api-key", _apiKey);

        var response = await
_httpClient.SendAsync(request);

        if (response.IsSuccessStatusCode)
        {
            return await
response.Content.ReadAsStringAsync();
        }
        else
        {
            return null;
        }
    }
}
```

Java

COPY

```
import java.io.IOException;
import java.net.HttpURLConnection;
import java.net.URL;

public class APIClient {
    private final String BASE_URL =
"https://api.example.com";
    private final String API_KEY = "YOUR_API_KEY";

    public String getData(String endpoint) throws
IOException {
        URL url = new URL(BASE_URL + "/" + endpoint);
        HttpURLConnection con = (HttpURLConnection)
url.openConnection();
        con.setRequestMethod("GET");
        con.setRequestProperty("x-api-key", API_KEY);

        int responseCode = con.getResponseCode();
        if (responseCode == HttpURLConnection.HTTP_OK) {
            // Read and return response
        } else {
            // Handle error
            return null;
        }
    }
}
```

Compliant Code Examples

.NET (C#)

COPY

```
using System;
using System.Net;
using System.Net.Http;
using System.Threading.Tasks;

public class APIClient
{
    private readonly HttpClient _httpClient;

    public APIClient()
    {
        _httpClient = new HttpClient();
    }

    public async Task<string> GetData(string endpoint,
string apiKey)
    {
        var request = new
HttpRequestMessage(HttpMethod.Get,
$"https://api.example.com/{endpoint}");
        request.Headers.Add("x-api-key", apiKey);

        var response = await
_httpClient.SendAsync(request);

        if (response.IsSuccessStatusCode)
        {
            return await
response.Content.ReadAsStringAsync();
        }
        else
        {
            return null;
```

```
        }
    }
}
```

Java

COPY

```
import java.io.IOException;
import java.net.HttpURLConnection;
import java.net.URL;

public class APIClient {
    private final String BASE_URL =
"https://api.example.com";

    public String getData(String endpoint, String apiKey)
throws IOException {
        URL url = new URL(BASE_URL + "/" + endpoint);
        HttpURLConnection con = (HttpURLConnection)
url.openConnection();
        con.setRequestMethod("GET");
        con.setRequestProperty("x-api-key", apiKey);

        int responseCode = con.getResponseCode();
        if (responseCode == HttpURLConnection.HTTP_OK) {
            // Read and return response
        } else {
            // Handle error
            return null;
        }
    }
}
```

The non-compliant code examples demonstrate hardcoded API keys, a common security issue in API development. Hardcoding keys makes them vulnerable to unauthorized access if the source code is exposed or compromised.

In the compliant code examples:

- The API key is passed as a parameter to the method instead of being hardcoded in the source code.
- This ensures that the API key is not directly visible in the code and can be provided securely during runtime.
- By removing hardcoded keys, the compliant code enhances security by reducing the risk of key exposure in case of source code leaks or unauthorized access.

Capturing and Inspecting JWTs

Before diving into specific attack techniques, it's essential to capture and inspect JWTs to understand their structure and potential vulnerabilities. Tools like Burp Suite Marketplace Extensions provide robust capabilities for capturing and analyzing JWTs in transit.

Common JWT Attack Techniques

1. Missing Verification

One prevalent attack scenario involves APIs failing to verify the integrity of JWTs properly. Developers may overlook the verification step, blindly trusting the JWT payload without verifying the signature. Attackers can exploit this oversight by tampering with the JWT payload, leading to unauthorized access.

2. None Attack

The None attack exploits instances where developers deliberately omit the signature stage during JWT creation by specifying the None parameter as the signature algorithm. Attackers can identify unsigned JWTs and manipulate their contents to gain unauthorized access to API endpoints.

3. Algorithm Switch Attack

This attack exploits misunderstandings or flawed implementations of JWT libraries regarding symmetric and asymmetric cryptography. Attackers trick the client into using a public key intended for asymmetric algorithms as the verification key for symmetric algorithms. By creating a malicious token signed with this key, attackers can bypass authentication mechanisms.

4. JWT Header Parameter Injections

JWTs specify header parameters like alg and typ, but other fields like jwk and jku can be manipulated by attackers. Techniques such as directory path traversal or key substitution attacks allow attackers to modify these parameters, potentially compromising API security.

5. Cracking JWTs

For symmetric algorithms like HS256, developers may use easily memorable passphrases instead of longer keys, making them susceptible to brute-force attacks. Tools like jwt_tool can apply wordlists against JWTs to identify the signature key, enabling attackers to forge new JWTs and gain unauthorized access.

Mitigation Strategies

To mitigate token-based attacks on APIs, consider the following strategies:

- **Implement Proper Verification:** Ensure APIs verify the integrity of JWTs before trusting their contents.
- **Avoid None Algorithm:** Refrain from using the None parameter as the signature algorithm to prevent unsigned JWT vulnerabilities.
- **Clarify Cryptographic Logic:** Educate developers on the differences between symmetric and asymmetric cryptography to prevent algorithm switch attacks.
- **Validate JWT Header Parameters:** Implement strict validation of JWT header parameters to prevent parameter injection attacks.
- **Enforce Strong Key Management:** Use strong cryptographic keys and enforce secure key management practices to protect against key cracking attacks.

Non-compliant Code Examples

.NET (C#)

COPY

```
using System;
using System.IdentityModel.Tokens.Jwt;

public class JWTHandler
{
    public void VerifyJWT(string token)
    {
```

```
        var handler = new JwtSecurityTokenHandler();
        var jwtToken = handler.ReadJwtToken(token);

        // Missing verification step
        // Trusting the token without verifying the
signature
        Console.WriteLine("JWT Payload: " +
jwtToken.Payload);
    }
}
```

Java

COPY

```
import io.jsonwebtoken.Claims;
import io.jsonwebtoken.Jws;
import io.jsonwebtoken.Jwts;

public class JWTHandler {
    public void verifyJWT(String token) {
        Jws<Claims> claimsJws =
Jwts.parser().parseClaimsJws(token);

        // Missing verification step
        // Trusting the token without verifying the
signature
        Claims body = claimsJws.getBody();
        System.out.println("JWT Payload: " + body);
    }
}
```

Compliant Code Examples

.NET (C#)

COPY

```
using System;
using System.IdentityModel.Tokens.Jwt;

public class JWTHandler
{
    public void VerifyJWT(string token)
    {
        var handler = new JwtSecurityTokenHandler();

        // Perform proper verification of the JWT
        var validationParameters = new
TokenValidationParameters
        {
            ValidateIssuerSigningKey = true,
            IssuerSigningKey = mySecurityKey,
            ValidateIssuer = true,
            ValidIssuer = "issuer",
            ValidateAudience = true,
            ValidAudience = "audience",
            ValidateLifetime = true
        };

        try
        {
            var claimsPrincipal =
handler.ValidateToken(token, validationParameters, out _);
            Console.WriteLine("JWT Payload: " +
claimsPrincipal.Claims);
        }
```

```
        catch (SecurityTokenException ex)
        {
            Console.WriteLine("JWT Validation Failed: " +
ex.Message);
        }
    }
}
```

Java

COPY

```
import io.jsonwebtoken.Claims;
import io.jsonwebtoken.Jws;
import io.jsonwebtoken.Jwts;

public class JWTHandler {
    public void verifyJWT(String token) {
        try {
            // Perform proper verification of the JWT
            Jws<Claims> claimsJws = Jwts.parser()
                    .setSigningKey(mySecretKey)
                    .parseClaimsJws(token);

            Claims body = claimsJws.getBody();
            System.out.println("JWT Payload: " + body);
        } catch (Exception ex) {
            System.out.println("JWT Validation Failed: " +
ex.getMessage());
        }
    }
}
```

In the non-compliant code examples, the JWT verification step is missing, leading to a vulnerability where the API blindly trusts the JWT payload without verifying its signature. This oversight allows attackers to tamper with the JWT payload and gain unauthorized access to API endpoints.

The compliant code examples address this vulnerability by implementing proper JWT verification mechanisms. They use libraries and methods provided by the respective platforms to validate the JWT signature, issuer, audience, and expiration time. This ensures that only valid and tamper-proof JWTs are accepted by the API, enhancing security against token-based attacks. Additionally, error handling is included to gracefully handle cases where JWT validation fails.

Weak Reset Process

A weak password reset process can be exploited by attackers to compromise user accounts. By analyzing the reset sequence, attackers can identify vulnerabilities such as predictable passwords or tokens, lack of rate-limiting, and insecure transmission of sensitive data. Let's explore potential attack vectors and mitigation measures:

Attack Vectors:

1. **User ID Enumeration:** Attackers may attempt to enumerate user IDs to initiate password resets for other accounts.
2. **Predictable Tokens:** Easily guessable or predictable tokens in reset URLs can be exploited to gain unauthorized access.

Mitigation Strategies:

- Implement strict validation checks for user-initiated password reset requests.
- Use strong, randomly generated tokens with a limited lifespan to prevent token guessing attacks.
- Enforce rate-limiting mechanisms to restrict the number of password reset attempts per user.

Insecure Transmission

Transmitting sensitive data over insecure channels can expose API communications to interception and data breaches. Common weaknesses include insecure transport, transmission of sensitive data in URLs, and weak encryption mechanisms. Let's address these vulnerabilities:

Vulnerabilities:

1. **Insecure Transport:** Lack of encrypted transport (TLS/SSL) exposes data to interception by malicious actors.
2. **Sensitive Data in URLs:** Transmitting sensitive information in URLs, especially in GET requests, poses a security risk.
3. **Weak Encryption:** Inadequate encryption mechanisms or exposure of encryption keys can compromise data confidentiality.

Mitigation Measures:

- Always use encrypted transport protocols (HTTPS) to secure API communications.
- Avoid transmitting sensitive data in URLs; instead, use POST requests with data sent in the request body.

- Employ strong encryption algorithms and secure key management practices to protect sensitive data.

Lack of Rate Limiting

Failure to implement effective rate-limiting mechanisms can expose APIs to brute-force attacks, where attackers attempt to guess credentials or crack encryption keys. Let's explore mitigation strategies to address this vulnerability:

Attack Scenario:

- Attackers may launch brute-force attacks to guess passwords, crack encryption keys, or conduct denial-of-service attacks.

Mitigation Techniques:

- Implement robust rate-limiting controls to restrict the number of requests from a single client within a specified time frame.
- Monitor API usage patterns and implement anomaly detection mechanisms to identify and mitigate suspicious activities.

Non-compliant Code Examples

.NET (C#)

COPY

```
using System;
using System.Net;
using System.Net.Mail;

public class PasswordResetHandler
```

```
{
    public void SendPasswordResetEmail(string userEmail)
    {
        // Send password reset email with insecure
transmission over SMTP
        using (var client = new
SmtpClient("smtp.example.com"))
        {
            client.UseDefaultCredentials = false;
            client.Credentials = new
NetworkCredential("username", "password");
            client.EnableSsl = false; // Insecure
transmission
            client.Port = 25;

            var mailMessage = new MailMessage
            {
                From = new
MailAddress("noreply@example.com"),
                Subject = "Password Reset",
                Body = "Click the link to reset your
password: http://example.com/reset?user=" + userEmail //
Sensitive data in URL
            };
            mailMessage.To.Add(userEmail);

            client.Send(mailMessage);
        }
    }
}
```

Java

COPY

```
import javax.mail.*;
import javax.mail.internet.InternetAddress;
import javax.mail.internet.MimeMessage;
import java.util.Properties;

public class PasswordResetHandler {
    public void sendPasswordResetEmail(String userEmail) {
        // Send password reset email with insecure
transmission over SMTP
        Properties properties = new Properties();
        properties.setProperty("mail.smtp.host",
"smtp.example.com");
        properties.setProperty("mail.smtp.port", "25");
        properties.setProperty("mail.smtp.auth", "true");

properties.setProperty("mail.smtp.starttls.enable",
"false"); // Insecure transmission

        Session session =
Session.getDefaultInstance(properties, new Authenticator()
{
            protected PasswordAuthentication
getPasswordAuthentication() {
                return new
PasswordAuthentication("username", "password");
            }
        });

        try {
            MimeMessage message = new
MimeMessage(session);
            message.setFrom(new
InternetAddress("noreply@example.com"));
```

```
            message.addRecipient(Message.RecipientType.TO,
new InternetAddress(userEmail));
            message.setSubject("Password Reset");
            message.setText("Click the link to reset your
password: http://example.com/reset?user=" + userEmail); //
Sensitive data in URL

            Transport.send(message);
        } catch (MessagingException ex) {
            ex.printStackTrace();
        }
    }
}
```

Compliant Code Examples

.NET (C#)

COPY

```
using System;
using System.Net;
using System.Net.Mail;

public class PasswordResetHandler
{
    public void SendPasswordResetEmail(string userEmail,
string resetToken)
    {
        using (var client = new
SmtpClient("smtp.example.com"))
        {
            client.UseDefaultCredentials = false;
            client.Credentials = new
```

```
NetworkCredential("username", "password");
            client.EnableSsl = true; // Enforce secure
transmission over TLS/SSL
            client.Port = 587;

            var mailMessage = new MailMessage
            {
                From = new
MailAddress("noreply@example.com"),
                Subject = "Password Reset",
                Body = "Click the link to reset your
password: http://example.com/reset?token=" + resetToken //
Use secure tokens in URL
            };
            mailMessage.To.Add(userEmail);

            client.Send(mailMessage);
        }
    }
}
```

Java

COPY

```
import javax.mail.*;
import javax.mail.internet.InternetAddress;
import javax.mail.internet.MimeMessage;
import java.util.Properties;

public class PasswordResetHandler {
    public void sendPasswordResetEmail(String userEmail,
String resetToken) {
        Properties properties = new Properties();
```

```
        properties.setProperty("mail.smtp.host",
"smtp.example.com");
        properties.setProperty("mail.smtp.port", "587");
        properties.setProperty("mail.smtp.auth", "true");

properties.setProperty("mail.smtp.starttls.enable",
"true"); // Enforce secure transmission over TLS/SSL

        Session session =
Session.getDefaultInstance(properties, new Authenticator()
{
            protected PasswordAuthentication
getPasswordAuthentication() {
                return new
PasswordAuthentication("username", "password");
            }
        });

        try {
            MimeMessage message = new
MimeMessage(session);
            message.setFrom(new
InternetAddress("noreply@example.com"));
            message.addRecipient(Message.RecipientType.TO,
new InternetAddress(userEmail));
            message.setSubject("Password Reset");
            message.setText("Click the link to reset your
password: http://example.com/reset?token=" + resetToken);
// Use secure tokens in URL

            Transport.send(message);
        } catch (MessagingException ex) {
            ex.printStackTrace();
        }
```

```
        }
    }
```

In the non-compliant code examples, the password reset process is vulnerable due to insecure transmission of sensitive data over SMTP without encryption (SSL/TLS). Additionally, sensitive information, such as reset tokens, is transmitted in the URL, making it susceptible to interception. These vulnerabilities expose the application to attacks such as eavesdropping and token theft.

The compliant code examples address these vulnerabilities by enforcing secure transmission over SMTP using TLS/SSL encryption. They also use randomly generated, secure tokens in the reset URL to prevent token guessing attacks. These measures enhance the security of the password reset process and mitigate the risk of unauthorized access to user accounts.

Object-Level Authorization (BOLA)

Broken Object-Level Authorization (BOLA) occurs when an API fails to properly enforce access controls, allowing unauthorized users to access specific objects or resources. Attackers can exploit BOLA vulnerabilities by following a systematic approach:

1. **Identify API Operations:** Look for operations that accept object IDs as parameters.
2. **Create Resources:** Create resources using one user (User A) and verify access.
3. **Attempt Unauthorized Access:** Using another user (User B) without access, attempt to access the same resource.

4. **Identify Vulnerable Endpoints:** If User B gains unauthorized access, it indicates a BOLA vulnerability.

Mitigation:

- Implement robust access control mechanisms to restrict access to specific objects based on user permissions.
- Regularly audit and review access control policies to ensure they align with security requirements.
- Utilize tools like Burp Suite to identify potential object IDs and vulnerable endpoints.

Function-Level Authorization (BFLA)

Broken Function-Level Authorization (BFLA) occurs when an API allows unauthorized users to execute privileged functions or actions. Attackers can exploit BFLA vulnerabilities by attempting to perform unauthorized actions using different user roles:

1. **Create Resource:** Perform an action as User A.
2. **Attempt Unauthorized Action:** Try to execute the same action as User B, who lacks the necessary privileges.
3. **Confirm Unauthorized Access:** If User B can execute the action, it indicates a BFLA vulnerability.

Mitigation:

- Implement granular access controls at the function level to enforce least privilege.

- Conduct thorough security testing to identify and remediate BFLA vulnerabilities.
- Regularly review and update access control configurations to align with evolving security requirements.

Non-Compliant Code Examples for BOLA (Broken Object-Level Authorization)

.NET (C#)

COPY

```
using System;
using System.Net;
using System.Net.Http;
using System.Threading.Tasks;

public class ResourceController
{
    private readonly HttpClient _httpClient;

    public ResourceController()
    {
        _httpClient = new HttpClient();
    }

    public async Task<string> GetResource(string
resourceId)
    {
        // Insecure: No authorization check
        var response = await
_httpClient.GetAsync($"https://api.example.com/resource/{r
esourceId}");
        return await response.Content.ReadAsStringAsync();
```

```
    }
}
```

Java

COPY

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.net.HttpURLConnection;
import java.net.URL;

public class ResourceController {
    public String getResource(String resourceId) throws
IOException {
        // Insecure: No authorization check
        URL url = new
URL("https://api.example.com/resource/" + resourceId);
        HttpURLConnection connection = (HttpURLConnection)
url.openConnection();
        connection.setRequestMethod("GET");

        int responseCode = connection.getResponseCode();
        if (responseCode == HttpURLConnection.HTTP_OK) {
            BufferedReader in = new BufferedReader(new
InputStreamReader(connection.getInputStream()));
            String inputLine;
            StringBuilder response = new StringBuilder();

            while ((inputLine = in.readLine()) != null) {
                response.append(inputLine);
            }
            in.close();
```

```
            return response.toString();
        }
        return null;
    }
}
```

The non-compliant code examples for Broken Object-Level Authorization (BOLA) lack proper authorization checks when retrieving resources. They directly fetch resources based on the provided resource ID without verifying the user's permissions. This oversight can lead to unauthorized access, where any user can retrieve resources regardless of their access rights.

Compliant Code Examples for BOLA

.NET (C#)

COPY

```
using System;
using System.Net;
using System.Net.Http;
using System.Threading.Tasks;

public class ResourceController
{
    private readonly HttpClient _httpClient;

    public ResourceController()
    {
        _httpClient = new HttpClient();
    }
```

```
    public async Task<string> GetResource(string
resourceId, string accessToken)
    {
        // Secure: Perform authorization check using
access token
        _httpClient.DefaultRequestHeaders.Authorization =
new
System.Net.Http.Headers.AuthenticationHeaderValue("Bearer"
, accessToken);
        var response = await
_httpClient.GetAsync($"https://api.example.com/resource/{r
esourceId}");

        if (response.StatusCode == HttpStatusCode.OK)
        {
            return await
response.Content.ReadAsStringAsync();
        }
        else
        {
            throw new
UnauthorizedAccessException("Unauthorized access");
        }
    }
}
```

Java

COPY

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.net.HttpURLConnection;
```

```
import java.net.URL;

public class ResourceController {
    public String getResource(String resourceId, String
accessToken) throws IOException {
        // Secure: Perform authorization check using
access token
        URL url = new
URL("https://api.example.com/resource/" + resourceId);
        HttpURLConnection connection = (HttpURLConnection)
url.openConnection();
        connection.setRequestMethod("GET");
        connection.setRequestProperty("Authorization",
"Bearer " + accessToken);

        int responseCode = connection.getResponseCode();
        if (responseCode == HttpURLConnection.HTTP_OK) {
            BufferedReader in = new BufferedReader(new
InputStreamReader(connection.getInputStream()));
            String inputLine;
            StringBuilder response = new StringBuilder();

            while ((inputLine = in.readLine()) != null) {
                response.append(inputLine);
            }
            in.close();
            return response.toString();
        } else if (responseCode ==
HttpURLConnection.HTTP_UNAUTHORIZED) {
            throw new SecurityException("Unauthorized
access");
        }
        return null;
```

```
        }
    }
```

The compliant code examples for Broken Object-Level Authorization (BOLA) include proper authorization checks before retrieving resources. They require an access token or credentials to be passed, and the API validates these tokens to ensure the user has the necessary permissions to access the requested resource. This mitigates the risk of unauthorized access and enforces proper object-level authorization.

Data Attacks

Data attacks involve unauthorized access to sensitive information exposed by APIs. These attacks can lead to data breaches and compromise user privacy. Two common types of data attacks include Excessive Information Exposure and Mass Assignment:

Excessive Information Exposure:

- Examine API responses for excessive information disclosure, such as sensitive data exposed in plain text.
- Utilize pagination schemes or verbose parameter options to extract large volumes of data from APIs.

Mass Assignment:

- Exploit APIs vulnerable to mass assignment by manipulating parameters to modify server-side data.
- Use tools like Arjun to automate parameter fuzzing and identify vulnerable endpoints.

- Gain insights into the technology stack used by the API to refine attack strategies.

Mitigation:

- Implement data minimization techniques to reduce the exposure of sensitive information.
- Encrypt sensitive data in transit and at rest to mitigate the risk of data interception.
- Conduct regular security assessments to identify and remediate vulnerabilities in data handling processes.

Non-Compliant Code Examples for Excessive Information Exposure

.NET (C#)

COPY

```
using System;
using System.Net.Http;
using System.Threading.Tasks;

public class DataController
{
    private readonly HttpClient _httpClient;

    public DataController()
    {
        _httpClient = new HttpClient();
    }

    public async Task<string> GetSensitiveData(string
```

```
userId)
    {
        // Insecure: Fetches sensitive data without proper
access control
        var response = await
_httpClient.GetAsync($"https://api.example.com/user/{userI
d}");
        return await response.Content.ReadAsStringAsync();
    }
}
```

Java

COPY

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.net.HttpURLConnection;
import java.net.URL;

public class DataController {
    public String getSensitiveData(String userId) throws
IOException {
        // Insecure: Fetches sensitive data without proper
access control
        URL url = new URL("https://api.example.com/user/"
+ userId);
        HttpURLConnection connection = (HttpURLConnection)
url.openConnection();
        connection.setRequestMethod("GET");

        int responseCode = connection.getResponseCode();
        if (responseCode == HttpURLConnection.HTTP_OK) {
```

```
            BufferedReader in = new BufferedReader(new
InputStreamReader(connection.getInputStream()));
            String inputLine;
            StringBuilder response = new StringBuilder();

            while ((inputLine = in.readLine()) != null) {
                response.append(inputLine);
            }
            in.close();
            return response.toString();
        }
        return null;
    }
}
```

The non-compliant code examples for Excessive Information Exposure fetch sensitive data without proper access control or encryption. They expose sensitive information directly in API responses, increasing the risk of unauthorized access and data breaches.

Compliant Code Examples for Excessive Information Exposure

.NET (C#)

COPY

```
using System;
using System.Net.Http;
using System.Threading.Tasks;

public class DataController
{
```

```
    private readonly HttpClient _httpClient;

    public DataController()
    {
        _httpClient = new HttpClient();
    }

    public async Task<string> GetSensitiveData(string
userId, string accessToken)
    {
        // Secure: Requires valid access token for
authorization
        _httpClient.DefaultRequestHeaders.Authorization =
new
System.Net.Http.Headers.AuthenticationHeaderValue("Bearer"
, accessToken);
        var response = await
_httpClient.GetAsync($"https://api.example.com/user/{userI
d}");

        if (response.IsSuccessStatusCode)
        {
            return await
response.Content.ReadAsStringAsync();
        }
        else
        {
            throw new
UnauthorizedAccessException("Unauthorized access");
        }
    }
}
```

Java

COPY

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.net.HttpURLConnection;
import java.net.URL;

public class DataController {
    public String getSensitiveData(String userId, String
accessToken) throws IOException {
        // Secure: Requires valid access token for
authorization
        URL url = new URL("https://api.example.com/user/"
+ userId);
        HttpURLConnection connection = (HttpURLConnection)
url.openConnection();
        connection.setRequestMethod("GET");
        connection.setRequestProperty("Authorization",
"Bearer " + accessToken);

        int responseCode = connection.getResponseCode();
        if (responseCode == HttpURLConnection.HTTP_OK) {
            BufferedReader in = new BufferedReader(new
InputStreamReader(connection.getInputStream()));
            String inputLine;
            StringBuilder response = new StringBuilder();

            while ((inputLine = in.readLine()) != null) {
                response.append(inputLine);
            }
            in.close();
            return response.toString();
```

```
        } else if (responseCode ==
HttpURLConnection.HTTP_UNAUTHORIZED) {
            throw new SecurityException("Unauthorized
access");
        }
        return null;
    }
}
```

The compliant code examples for Excessive Information Exposure implement proper access control and encryption measures to protect sensitive data. They require valid access tokens for authorization and encrypt sensitive data in transit, reducing the risk of unauthorized access and data exposure.

Detecting Injection Vulnerabilities

The fundamental concept of injection attacks is straightforward: inject malicious payloads into input fields and observe the system's response. Detection of successful injection attempts can vary, often resulting in crashes or generic error messages. Understanding these responses is crucial for identifying vulnerabilities.

Common Injection Points:

- **Query Strings:** Manipulating parameters in the URL query string.
- **Tokens:** Exploiting vulnerabilities in token-based authentication mechanisms.
- **Parameters in Requests:** Injecting payloads into parameters of POST or PUT requests.

- **Headers:** Manipulating headers to execute malicious actions.

Detection Techniques:

- **Error Messages:** Analyze error messages for clues indicating successful injection attempts.
- **Behavioral Analysis:** Observe changes in system behavior or responses to injected payloads.
- **Manual Testing:** Conduct manual testing using crafted payloads to identify vulnerable endpoints.

SQL Injection

SQL injection is perhaps the most well-known injection attack, exploiting vulnerabilities in SQL queries to execute unauthorized database operations. By injecting malicious SQL code into input fields, attackers can manipulate query logic and gain unauthorized access to database contents.

```
-- Example URL: https://insecure-website.com/products?category=Gifts
-- SQL Injection Payload: ' OR 1=1--
```

This payload modifies the underlying SQL query to return all records, effectively bypassing authentication or authorization checks.

Mitigation:

- Use parameterized queries or prepared statements to prevent dynamic query construction.
- Implement input validation and sanitization to filter out malicious input.

- Regularly update and patch database systems to address known vulnerabilities.

NoSQL Injection

NoSQL databases, like MongoDB, are also susceptible to injection attacks, similar to SQL databases. Attackers can exploit vulnerabilities in NoSQL queries to execute unauthorized operations and access sensitive data.

```
// Example Payload: { "username": {"$ne": null}, "password": {"$ne": null} }
```

This payload manipulates the query to bypass authentication checks and gain unauthorized access.

Mitigation:

- Utilize parameterized queries and ORM frameworks to abstract database interactions.
- Implement strict input validation to prevent injection attacks.
- Regularly review and audit query logic to identify and remediate vulnerabilities.

Command Injection

Command injection allows attackers to execute arbitrary commands on the underlying operating system, posing a severe security risk. By injecting malicious commands into input fields, attackers can gain unauthorized access to system resources and execute malicious operations.

```
# Example Command: ping -c 1 192.168.0.1; ls -la
```

This payload appends a command to a legitimate operation, allowing attackers to execute arbitrary commands.

Mitigation:

- Avoid executing user-supplied input as system commands.
- Implement input validation and sanitization to filter out malicious characters.
- Use application-level firewalls and security mechanisms to detect and block command injection attempts.

Path Traversal

Path traversal attacks exploit vulnerabilities in file system access mechanisms, allowing attackers to access files or directories outside of the intended scope. By manipulating file paths in input fields, attackers can bypass access controls and access sensitive system files.

```
# Example Payload: ../../../../../etc/passwd
```

This payload traverses directory structures to access sensitive system files, such as the passwd file.

Mitigation:

- Implement strict file path validation and access controls.
- Use whitelisting to restrict file system access to authorized directories.

- Regularly monitor and audit file system access logs for suspicious activities.

Server-Side Request Forgery (SSRF)

SSRF attacks allow attackers to trick the server into making unauthorized requests to internal or external resources. By manipulating input fields, attackers can force the server to access malicious URLs under their control, leading to data exfiltration or server compromise.

```
# Example Request: POST /product/stock HTTP/1.0 Content-Type:
application/x-www-form-urlencoded basketApi= http://localhost/admin
```

This payload deceives the server into accessing a rogue URL, potentially leading to server compromise.

Mitigation:

- Implement strict input validation to prevent SSRF attacks.
- Use whitelisting to restrict access to trusted internal and external resources.
- Regularly update and patch server-side components to address known vulnerabilities.

Non-Compliant Code Example for SQL Injection (Java)

COPY

```
import java.sql.Connection;
import java.sql.DriverManager;
```

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

public class ProductDAO {
    private static final String DB_URL =
"jdbc:mysql://localhost:3306/mydatabase";
    private static final String DB_USER = "username";
    private static final String DB_PASSWORD = "password";

    public String getProductByCategory(String category) {
        String query = "SELECT * FROM products WHERE
category = '" + category + "'";
        StringBuilder result = new StringBuilder();

        try (Connection conn =
DriverManager.getConnection(DB_URL, DB_USER, DB_PASSWORD);
             Statement stmt = conn.createStatement();
             ResultSet rs = stmt.executeQuery(query)) {

            while (rs.next()) {
                result.append("Name:
").append(rs.getString("name")).append(", ")
                      .append("Price:
").append(rs.getString("price")).append("\n");
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }
        return result.toString();
    }
}
```

The non-compliant code directly concatenates the `category` parameter into the SQL query string, making it vulnerable to SQL injection attacks. Attackers can manipulate the `category` parameter to inject malicious SQL code and gain unauthorized access to the database.

Compliant Code Example for SQL Injection (Java)

COPY

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

public class ProductDAO {
    private static final String DB_URL =
"jdbc:mysql://localhost:3306/mydatabase";
    private static final String DB_USER = "username";
    private static final String DB_PASSWORD = "password";

    public String getProductByCategory(String category) {
        String query = "SELECT * FROM products WHERE
category = ?";
        StringBuilder result = new StringBuilder();

        try (Connection conn =
DriverManager.getConnection(DB_URL, DB_USER, DB_PASSWORD);
             PreparedStatement pstmt =
conn.prepareStatement(query)) {

            pstmt.setString(1, category);
            try (ResultSet rs = pstmt.executeQuery()) {
                while (rs.next()) {
```

```
                    result.append("Name:
").append(rs.getString("name")).append(", ")
                            .append("Price:
").append(rs.getString("price")).append("\n");
                }
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }
        return result.toString();
    }
}
```

The compliant code uses a parameterized SQL query to prevent SQL injection attacks. Instead of concatenating user input directly into the query string, it uses a placeholder (`?`) and sets the parameter value using `setString()` method, ensuring proper escaping and sanitization of user input. This mitigates the risk of SQL injection vulnerabilities.

Non-Compliant Code Example for Injection Vulnerabilities (Java)

COPY

```
import java.net.HttpURLConnection;
import java.net.URL;
import java.io.BufferedReader;
import java.io.InputStreamReader;

public class APIClient {
    public String fetchData(String url) {
        StringBuilder response = new StringBuilder();
```

```
        try {
            URL apiUrl = new URL(url);
            HttpURLConnection conn = (HttpURLConnection)
apiUrl.openConnection();
            conn.setRequestMethod("GET");
            conn.connect();

            BufferedReader in = new BufferedReader(new
InputStreamReader(conn.getInputStream()));
            String inputLine;

            while ((inputLine = in.readLine()) != null) {
                response.append(inputLine);
            }
            in.close();
        } catch (Exception e) {
            e.printStackTrace();
        }

        return response.toString();
    }
}
```

The non-compliant Java code retrieves data from a given URL using the GET method without proper input validation or sanitization. Attackers can exploit this code by manipulating the URL query string, injecting malicious payloads, and potentially compromising the system.

Compliant Code Example for Injection Vulnerabilities (Java)

COPY

```
import java.net.HttpURLConnection;
import java.net.URL;
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.UnsupportedEncodingException;
import java.net.URLEncoder;

public class APIClient {
    public String fetchData(String url) {
        StringBuilder response = new StringBuilder();

        try {
            // Encode URL to prevent injection attacks
            String encodedUrl = URLEncoder.encode(url,
"UTF-8");
            URL apiUrl = new URL(encodedUrl);

            HttpURLConnection conn = (HttpURLConnection)
apiUrl.openConnection();
            conn.setRequestMethod("GET");
            conn.connect();

            BufferedReader in = new BufferedReader(new
InputStreamReader(conn.getInputStream()));
            String inputLine;

            while ((inputLine = in.readLine()) != null) {
                response.append(inputLine);
            }
            in.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
```

```
            return response.toString();
        }
    }
```

The compliant Java code properly encodes the URL using `URLEncoder.encode()` to prevent injection attacks via the URL query string. This ensures that any special characters in the URL are encoded, mitigating the risk of injection vulnerabilities.

API Abuse

API abuse involves exploiting APIs in ways not intended by their designers, often resulting in excessive data exfiltration or misuse of underlying business processes. This type of attack does not necessarily rely on a specific vulnerability in the API itself but rather on exploiting its functionalities beyond their intended scope.

Example:

- Scraping data from airline booking sites or real estate agencies by excessively querying their APIs.
- Using APIs in ways not possible through the website interface, such as bypassing restrictions or manipulating data flows.

Unrestricted Access to Sensitive Business Flows

This vulnerability category, introduced in the OWASP API Security Top 10 in 2023, focuses on exploiting the logic of underlying business flows enabled by APIs. Attackers leverage APIs to manipulate business

processes, often resulting in unauthorized access to sensitive resources or services.

Example:

- Exploiting an online booking system to bulk book or reserve ticketed items without payment.
- Operating bots on ticket scalping sites to exploit scarcity and profit from high-demand events.

Business Logic Attacks

Business logic attacks target APIs with fragile implementations that rely heavily on client adherence to prescribed workflows. Attackers exploit vulnerabilities in business logic, such as arbitrary restrictions or specific input format requirements, to disrupt normal operations or extract sensitive information.

Example:

- Ignoring setup API calls to expose default behaviors or leak internal information.
- Submitting data in unexpected formats to trigger erratic behavior and uncover underlying weaknesses.

Non-Compliant .NET Code for API Abuse

COPY

```
using System;
using System.Net.Http;
```

```
class Program
{
    static void Main()
    {
        string apiUrl = "https://example.com/api/data";

        for (int i = 0; i < 1000; i++)
        {
            HttpClient client = new HttpClient();
            HttpResponseMessage response =
client.GetAsync(apiUrl).Result;

            if (response.IsSuccessStatusCode)
            {
                string data =
response.Content.ReadAsStringAsync().Result;
                Console.WriteLine(data);
            }
        }
    }
}
```

The non-compliant .NET code excessively queries an API endpoint (`https://example.com/api/data`) within a loop without proper rate-limiting or throttling mechanisms. This can lead to API abuse, overwhelming the server and potentially disrupting its normal operation.

Compliant .NET Code for API Abuse

COPY

```
using System;
using System.Net.Http;
using System.Threading.Tasks;

class Program
{
    static async Task Main()
    {
        string apiUrl = "https://example.com/api/data";

        for (int i = 0; i < 10; i++) // Limit requests to
10
        {
            HttpClient client = new HttpClient();
            HttpResponseMessage response = await
client.GetAsync(apiUrl);

            if (response.IsSuccessStatusCode)
            {
                string data = await
response.Content.ReadAsStringAsync();
                Console.WriteLine(data);
            }
        }
    }
}
```

The compliant .NET code limits the number of requests to the API endpoint (`https://example.com/api/data`) to 10 within a loop. Additionally, it utilizes asynchronous programming to improve performance and responsiveness. This implementation helps prevent API abuse by imposing a request limit.

Non-Compliant Java Code for API Abuse

COPY

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.net.HttpURLConnection;
import java.net.URL;

public class APIClient {
    public static void main(String[] args) {
        String apiUrl = "https://example.com/api/data";

        for (int i = 0; i < 1000; i++) {
            try {
                URL url = new URL(apiUrl);
                HttpURLConnection conn =
(HttpURLConnection) url.openConnection();
                conn.setRequestMethod("GET");
                conn.connect();

                BufferedReader in = new BufferedReader(new
InputStreamReader(conn.getInputStream()));
                String inputLine;
                StringBuffer response = new
StringBuffer();

                while ((inputLine = in.readLine()) !=
null) {
                    response.append(inputLine);
                }
                in.close();

                System.out.println(response.toString());
            } catch (Exception e) {
```

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

The non-compliant Java code excessively queries an API endpoint (`https://example.com/api/data`) within a loop without implementing any rate-limiting or throttling mechanisms. This can lead to API abuse, causing server overload and potential disruption of service.

Compliant Java Code for API Abuse

COPY

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.net.HttpURLConnection;
import java.net.URL;

public class APIClient {
    public static void main(String[] args) {
        String apiUrl = "https://example.com/api/data";

        for (int i = 0; i < 10; i++) { // Limit requests
to 10
            try {
                URL url = new URL(apiUrl);
                HttpURLConnection conn =
(HttpURLConnection) url.openConnection();
                conn.setRequestMethod("GET");
                conn.connect();
```

```
                BufferedReader in = new BufferedReader(new
InputStreamReader(conn.getInputStream()));
                String inputLine;
                StringBuffer response = new
StringBuffer();

                while ((inputLine = in.readLine()) !=
null) {
                    response.append(inputLine);
                }
                in.close();

                System.out.println(response.toString());
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }
}
```

The compliant Java code limits the number of requests to the API endpoint (`https://example.com/api/data`) to 10 within a loop. By adding this restriction, it prevents API abuse and potential server overload. Additionally, it handles exceptions gracefully to avoid service disruptions.

Resources

- Defending APIs by Colin Domoney