AYI NEDJIMI
Consultants

Reverse Engineering .NET : Décompilation, Analyse et

Catégorie : Retro-Ingenierie Lecture : 9 min Publié le : 08/03/2026 Auteur : Ayi NEDJIMI

Guide complet de reverse engineering .NET : décompilation avec dnSpy et ILSpy, analyse IL/MSIL, debugging, déobfuscation ConfuserEx, analyse malware.

Avertissement : Les techniques présentées dans cet article sont destinées exclusivement à des fins éducatives et de tests autorisés. Toute utilisation malveillante est illégale et contraire à l'éthique professionnelle.

2.1 Le Common Language Runtime (CLR)

Le CLR est le moteur d'exécution de .NET, l'équivalent de la JVM pour Java. Il fournit un environnement managé qui gère la mémoire (garbage collector), la sécurité (Code Access Security, désormais déprécié dans .NET Core), le chargement des types et la compilation Just-In-Time (JIT). Comprendre le CLR est fondamental pour le reverse engineering car c'est lui qui transforme le bytecode IL en code natif exécutable. Guide complet de reverse engineering .NET : décompilation avec dnSpy et ILSpy, analyse IL/MSIL, debugging, déobfuscation ConfuserEx, analyse malware. La rétro-ingénierie est une discipline fondamentale en analyse de malware et en recherche de vulnérabilités. Reverse Engineering .NET : Décompilation, Analyse et couvre les techniques avancées utilisées par les analystes. Nous abordons notamment : 9. protection du code .net, 10. checklist d'analyse .net et questions frequentes. Les professionnels y trouveront des recommandations actionnables, des commandes prêtes à l'emploi et des stratégies de mise en œuvre adaptées aux environnements d'entreprise.

L'architecture du CLR se décompose en plusieurs composants critiques pour l'analyste :

- **Class Loader** : charge les assemblies et résout les dépendances. Point d'interception pour les techniques d'injection (Assembly.Load, Assembly.LoadFrom).
- **JIT Compiler (RyuJIT)** : compile le code IL en code natif à la volée, méthode par méthode. Le code natif généré peut être capturé avec WinDbg + SOS pour une analyse post-JIT.
- **Garbage Collector (GC)** : gestion automatique de la mémoire avec trois générations. Les malwares .NET peuvent manipuler le GC pour éviter la détection en mémoire.
- **Type System** : définit la hiérarchie des types (classes, interfaces, structs, enums). Les métadonnées de types sont la clé de la décompilation réussie.
- **Exception Handler** : gestion structurée des exceptions (SEH managé). Les obfuscateurs utilisent intensivement les blocs try/catch/finally pour complexifier le control flow.

2.2 Intermediate Language (IL/MSIL/CIL)

L'Intermediate Language -- appelé IL, MSIL (Microsoft Intermediate Language) ou CIL (Common Intermediate Language) -- est un bytecode stack-based, conceptuellement similaire au bytecode Java mais avec des spécificités propres à .NET. Chaque instruction IL manipule une pile d'évaluation (evaluation stack) : les opérandes sont poussés sur la pile, les opérations consomment les éléments du sommet et poussent le résultat.

Les instructions IL les plus fréquemment rencontrées en analyse de malwares :

```
// Instructions de chargement (Load)
ldstr "http://c2server.evil.com"    // Charge une chaîne sur la pile
ldarg.0                             // Charge le premier argument (this pour les méthodes
d'instance)
ldc.i4 0x1337                       // Charge un entier 32 bits
ldsfld class Malware.Config::Key    // Charge un champ statique

// Instructions d'appel
call void [System.Net.Http]System.Net.Http.HttpClient::GetAsync(string)
callvirt instance string [mscorlib]System.IO.StreamReader::ReadToEnd()
newobj instance void Malware.Payload::.ctor()

// Instructions de contrôle de flux
br.s IL_0042          // Branch inconditionnelle (jump)
brfalse.s IL_0028     // Branch si false/null/0
brtrue IL_005A        // Branch si true/non-null/non-0
ret                   // Return

// Instructions de stockage
stloc.0               // Stocke le sommet de la pile dans la variable locale 0
stfld string Malware.Config::C2Url  // Stocke dans un champ d'instance
```

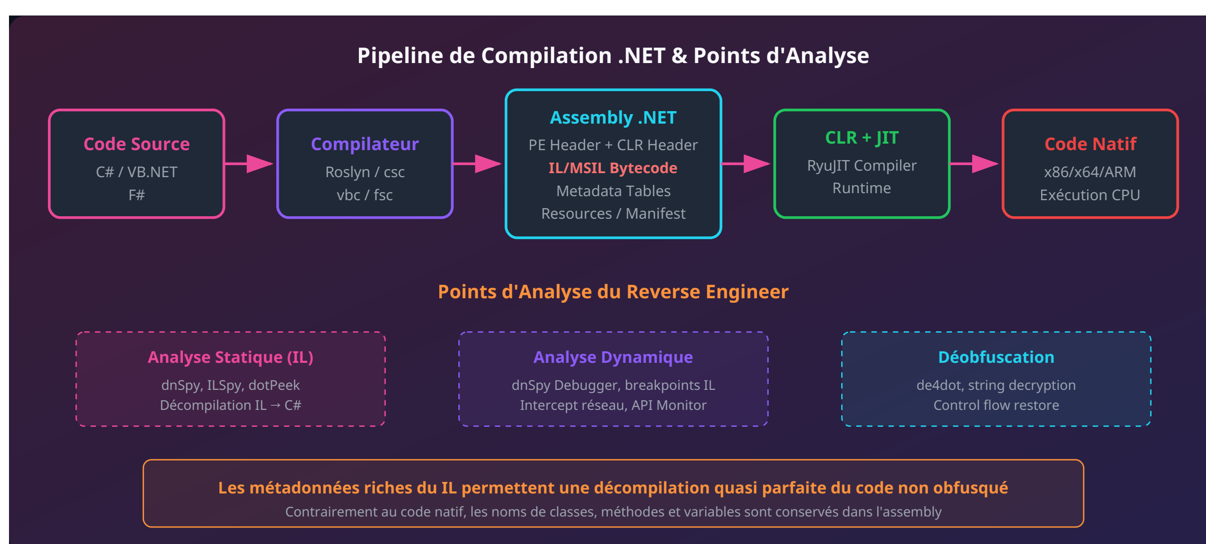
Une particularité essentielle du IL pour le reverse engineering : chaque instruction référence des **metadata tokens** qui pointent vers les tables de métadonnées de l'assembly. Cela signifie que même sans symboles de debug, on dispose des noms de types, méthodes, champs et paramètres -- un luxe inexistant en analyse de binaires natifs. C'est pourquoi les obfuscateurs .NET se concentrent en priorité sur le renommage de ces éléments.

2.3 Structure des assemblies .NET (PE + métadonnées)

Un assembly .NET est un fichier PE (Portable Executable) augmenté de structures spécifiques au CLR. L'en-tête PE standard contient un répertoire **CLR Header** (aussi appelé COM+ Runtime Header, à l'offset DataDirectory[14]) qui pointe vers les métadonnées .NET. Cette structure est critique car c'est elle que les outils de décompilation analysent en premier.

Les tables de métadonnées (**Metadata Tables**) constituent le coeur de l'assembly et contiennent :

Table	Contenu	Intérêt pour le RE
TypeDef	Définitions de types (classes, structs)	Structure du programme, hiérarchie de classes
MethodDef	Définitions de méthodes + RVA du code IL	Points d'entrée, fonctions critiques
Field	Champs des types	Configurations, clés, URLs
MemberRef	Références vers des membres externes	APIs utilisées (réseau, crypto, fichiers)
AssemblyRef	Assemblies référencées	Dépendances, frameworks utilisés
UserString	Chaînes littérales du code	URLs C2, clés de chiffrement, messages
StandAloneSig	Signatures de variables locales	Compréhension des fonctions
CustomAttribute	Attributs personnalisés	Obfuscation markers, Confuser tags



Notre avis d'expert

Les techniques d'anti-analyse deviennent de plus en plus sophistiquées. Les packers, le code polymorphe et les checks d'environnement virtuel compliquent considérablement le travail d'analyse. La maîtrise des outils de désobfuscation est devenue indispensable.

JetBrains dotPeek est un décompilateur gratuit qui excelle dans la génération de projets Visual Studio reconstituables à partir d'assemblies. Sa force réside dans la qualité de la décompilation et la capacité à servir de **symbol server**, permettant de débugger du code tiers dans Visual Studio comme s'il disposait des sources originales.

de4dot est l'outil de déobfuscation .NET le plus connu. Bien que son développement soit ralenti, il reste efficace contre la majorité des obfuscateurs commerciaux et détecte automatiquement le type d'obfuscation utilisé :

```
# Détecter le type d'obfuscation
de4dot --detect malware.exe
# Output: Detected ConfuserEx v1.0.0

# Déobfusquer automatiquement
de4dot malware.exe -o malware-clean.exe

# Déobfuscation avec options spécifiques
de4dot malware.exe --strtyp delegate --strtok 0x06000123 -o clean.exe

# Pipeline complet d'analyse
de4dot malware.exe -o step1.exe && ilspycmd step1.exe -o ./decompiled -p
```

Outil	Décompilation	Debugging	Déobfuscation	Édition IL	CLI	Prix
dnSpy/dnSpyEx	Excellente	Oui (intégré)	Non	Oui	Non	Gratuit (GPL)
ILSpy	Excellente	Non	Non	Non	Oui	Gratuit (MIT)
dotPeek	Très bonne	Via VS	Non	Non	Non	Gratuit
de4dot	Non	Non	Oui (auto)	Non	Oui	Gratuit (GPL)
Telerik JustDecompile	Bonne	Non	Non	Non	Non	Gratuit
.NET Reflector	Excellente	Via VS	Non	Plugin	Non	Payant (~95$)

Cas concret

L'analyse du wiper HermeticWiper, déployé contre des organisations ukrainiennes en février 2022, a révélé l'utilisation d'un driver légitime de partitionnement pour corrompre le MBR et les partitions NTFS. La rétro-ingénierie rapide par ESET et SentinelOne a permis de publier des indicateurs de compromission en moins de 24 heures.

```
// Exemple de structure typique d'un malware .NET
// Namespace principal souvent obfusqué
namespace a8f3e2
{
    // Classe principale avec entry point
    internal static class b7c1d9
    {
        // Entry point - Main obfusqué
        private static void c4a2e1(string[] args)
        {
            // Déchiffrement de la configuration
            string config = d5b3f2.e6c4a3("base64encodedstring==");

            // Initialisation du C2
            f7d5b4 client = new f7d5b4(config);
            client.g8e6c5(); // Boucle principale
        }
    }

    // Classe de déchiffrement des chaînes
    internal static class d5b3f2
    {
        internal static string e6c4a3(string input)
        {
            byte[] data = Convert.FromBase64String(input);
            // XOR ou AES déchiffrement
            return Encoding.UTF8.GetString(Decrypt(data));
        }
    }
}
```

4.3 Identification des entry points et du flux d'exécution

L'identification du flux d'exécution commence par le point d'entrée managé, mais les malwares .NET modernes utilisent plusieurs techniques pour complexifier ce flux :

- **Module Initializers (.cctor)** : le constructeur statique du module (`<Module>.cctor`) s'exécute avant Main. Les malwares l'utilisent pour l'anti-debug ou le déchiffrement précoce.
- **Assembly Resolve Events** : handlers `AppDomain.AssemblyResolve` qui chargent dynamiquement des assemblies depuis les ressources ou le réseau.
- **Reflection Loading** : `Assembly.Load(byte[])` pour charger des assemblies en mémoire sans les écrire sur disque.
- **Dynamic Method Invocation** : `MethodInfo.Invoke()` pour appeler des méthodes par réflexion, cachant le vrai flux d'exécution.

Astuce d'analyste : tracer les Assembly.Load

Placez un breakpoint sur `System.Reflection.Assembly.Load(byte[])` dans dnSpy. Quand le breakpoint est atteint, examinez le tableau d'octets sur la pile -- c'est souvent le payload réel du malware. Sauvegardez ces octets et ouvrez-les comme un nouvel assembly dans dnSpy pour continuer l'analyse. Cette technique est documentée dans notre guide sur la déobfuscation de malwares polymorphes.

Les **proxy calls** (ou call hiding) remplacent les appels de méthodes directs par des appels indirects via des delegates. Au lieu d'un `call System.Net.WebClient::DownloadString` visible dans le IL, l'obfuscateur génère un delegate initialisé dynamiquement qui pointe vers la méthode cible. Cela cache les imports et empêche l'analyse statique de déterminer quelles APIs sont utilisées.

```
// Avant obfuscation (call direct visible)
IL_0010: callvirt instance string [System]System.Net.WebClient::DownloadString(string)

// Après proxy call obfuscation
// Un delegate est initialisé dans le .cctor avec la méthode cible
// L'appel devient indirect :
IL_0010: ldsfld class [mscorlib]System.Func`2<string,string> ProxyClass::delegate_42
IL_0015: ldarg.1
IL_0016: callvirt instance !1 class [mscorlib]System.Func`2<string,string>::Invoke(!0)

// Résolution : identifier le .cctor, extraire les initialisations de delegates
// de4dot résout la plupart des proxy calls automatiquement
```

Malwares avec obfuscation custom

Les malwares les plus avancés (APT, ransomware enterprise) utilisent des forks custom de ConfuserEx ou des obfuscateurs internes non reconnus par de4dot. Dans ces cas, l'analyse manuelle avec dnSpy reste la seule option. Pour les techniques avancées d'anti-analyse, voir notre article sur les techniques anti-rétro-ingénierie des APT.

Points d'analyse clés :

- **Configuration** : stockée dans une classe `Settings` avec le host C2, le port, le mutex, le certificat TLS et la clé AES pour le chiffrement.
- **Persistance** : clés de registre Run, tâches planifiées, ou copie dans le dossier Startup. Ces techniques rejoignent celles documentées dans notre article sur la persistence multi-plateforme.
- **Anti-Analysis** : vérification de sandbox (nombre de CPU, RAM, présence de drivers VM), delayed execution (Thread.Sleep de 10-30 secondes).
- **Communication** : protocole binaire custom sur TCP avec sérialisation MessagePack et chiffrement AES-256-CBC.

8.3 NjRAT (Bladabindi) : persistance et propagation

NjRAT est l'un des RAT .NET les plus anciens et les plus persistants dans le domaine des menaces, actif depuis 2013. Malgré son ancienneté, il reste largement utilisé, notamment au Moyen-Orient et en Afrique du Nord. Son code source est publiquement disponible, ce qui a engendré des centaines de variantes.

Signatures d'analyse pour NjRAT :

```
// Indicateurs typiques de NjRAT dans le code décompilé

// 1. Mutex format caractéristique
string mutex = "d3d9c3b2a1";  // Souvent un hash court

// 2. Registry keys de persistance
Registry.CurrentUser.OpenSubKey("Software\\Microsoft\\Windows\\CurrentVersion\\Run", true)
    .SetValue("WindowsService", Application.ExecutablePath);

// 3. Communication C2 avec séparateur pipe "|'|"
string packet = "inf" + "|'|" + computerName + "|'|" + userName + "|'|" + osVersion;

// 4. Keylogger basique avec GetAsyncKeyState
[DllImport("user32.dll")]
static extern short GetAsyncKeyState(int vKey);

// 5. Spread via USB (copie + autorun.inf)
foreach (DriveInfo drive in DriveInfo.GetDrives())
    if (drive.DriveType == DriveType.Removable)
        File.Copy(Application.ExecutablePath, drive.Name + "\\system.exe");
```

Extraction automatique des IOCs

Pour les malwares .NET courants (AgentTesla, AsyncRAT, NjRAT, RedLine, Quasar), des outils d'extraction automatique de configuration existent : **CAPE Sandbox** (extraction auto via modules Python), **MalwareConfig** (parsers dédiés par famille) et les frameworks d'analyse assistés par IA. Ces outils extraient directement les IOCs (C2, credentials, clés crypto) sans nécessiter de reverse engineering manuel.

9. Protection du code .NET

9.1 Compilation AOT et NativeAOT

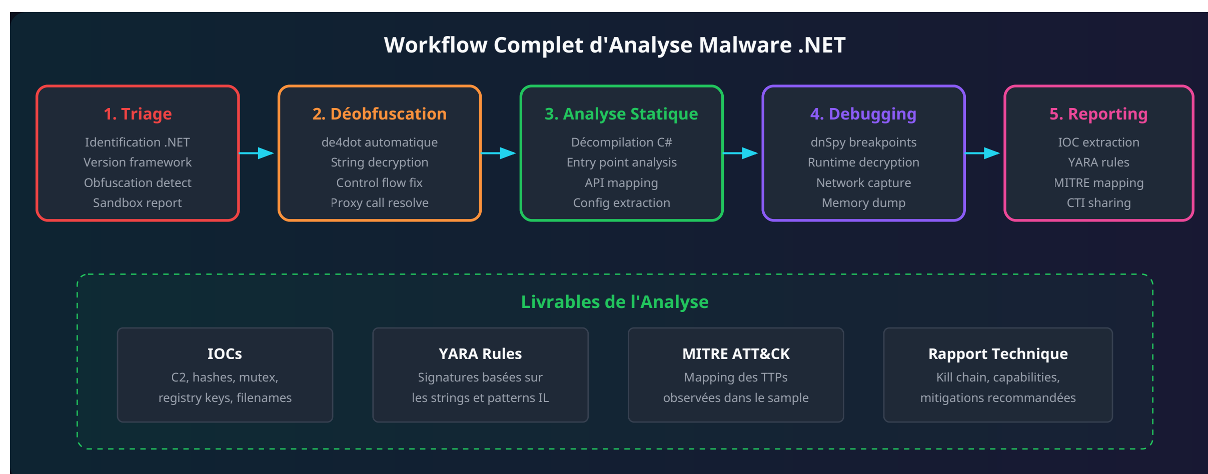
La compilation **Ahead-of-Time (AOT)** représente le changement de approche le plus significatif pour la protection du code .NET. Avec **.NET NativeAOT** (disponible depuis .NET 7, mature dans .NET 8+), le code C# est compilé directement en code natif sans inclure le runtime CLR ni le code IL dans le binaire final. Cela élimine fondamentalement le vecteur de décompilation IL.

```
# Compilation NativeAOT avec .NET 8
dotnet publish -c Release -r win-x64 --self-contained \
  /p:PublishAot=true \
  /p:StripSymbols=true \
  /p:OptimizationPreference=Size

# Le binaire résultant est du code natif x64 pur
# Plus de métadonnées .NET, plus de IL, plus de décompilation C#
# L'analyse requiert IDA Pro / Ghidra comme pour tout binaire natif

# Limitations NativeAOT :
# - Pas de reflection dynamique (Assembly.Load impossible)
# - Pas de dynamic code generation (Expression trees limités)
# - Taille du binaire plus importante (runtime embarqué en natif)
# - Temps de compilation significativement plus long
```

9.2 Stratégies de protection multi-couches

Pour les applications qui ne peuvent pas utiliser NativeAOT (dépendances sur la reflection, plugins dynamiques, etc.), une approche de **défense en profondeur** combinant plusieurs techniques est recommandée :

- **Obfuscation commerciale** : utiliser un obfuscateur de qualité (.NET Reactor avec NecroBit, Agile.NET avec virtualisation) comme première couche.
- **Code splitting** : séparer la logique sensible dans des bibliothèques natives (C/C++ via P/Invoke ou C++/CLI) pour les algorithmes critiques.
- **Vérification d'intégrité** : implémenter des checks de hash sur les assemblies au runtime, indépendamment de ceux de l'obfuscateur.
- **Licensing côté serveur** : ne jamais embarquer les clés de licence ou la logique de validation dans le binaire. Utiliser une validation côté serveur.
- **Heartbeat et revocation** : call-home périodique pour vérifier la validité de la licence et la possibilité de révoquer des instances compromises.



Pour approfondir ce sujet, consultez notre outil open-source malware-analysis-toolkit qui facilite l'analyse automatisée de malwares.

10. Checklist d'analyse .NET

Cette checklist synthétise le workflow complet d'analyse d'un binaire .NET suspect. Elle est utilisable comme aide-mémoire lors d'investigations et complète notre guide général de reverse engineering et analyse malware.

Checklist complète d'analyse .NET

- **Phase 1 - Triage**
  - Confirmer le format .NET (CLR Header, mscoree.dll import)
  - Identifier la version du framework (.NET Framework 4.x, .NET 6/7/8, Mono)
  - Scanner avec VirusTotal, ANY.RUN pour un rapport initial
  - Calculer les hashes (MD5, SHA1, SHA256) et les soumettre à la CTI
  - Exécuter en sandbox (CAPE, Joe Sandbox) pour les comportements réseau

- **Phase 2 - Déobfuscation**
  - Détecter l'obfuscateur avec de4dot ou par inspection des attributs
  - Appliquer de4dot avec les options adaptées à l'obfuscateur détecté
  - Vérifier le résultat en ouvrant dans ILSpy -- le code doit être lisible
  - Si de4dot échoue, tenter une déobfuscation manuelle des chaînes
- **Phase 3 - Analyse statique**
  - Ouvrir dans dnSpy, localiser le point d'entrée (Main ou .cctor)
  - Mapper les imports : réseau (WebClient, HttpClient), crypto (AES, RSA), fichiers, registre
  - Identifier les ressources embarquées (payloads chiffrés, configs)
  - Documenter les chaînes significatives (URLs, chemins, clés)
  - Tracer le flux d'exécution du Main vers les fonctions clés
- **Phase 4 - Analyse dynamique**
  - Configurer dnSpy debugger avec anti-anti-debug activé
  - Placer des breakpoints sur Assembly.Load, WebClient, Process.Start
  - Capturer les chaînes déchiffrées au runtime via tracepoints
  - Extraire les payloads en mémoire (dump des byte[] après déchiffrement)
  - Capturer le trafic réseau avec Wireshark/Fiddler pendant l'exécution
- **Phase 5 - IOC extraction et reporting**
  - Extraire tous les IOCs : C2 URLs, IPs, domains, hashes de payloads secondaires
  - Créer des règles YARA basées sur les chaînes uniques et les patterns de code
  - Mapper les TTPs observées sur le framework MITRE ATT&CK
  - Rédiger le rapport technique avec kill chain et recommandations de détection

Pour approfondir, consultez les ressources de MITRE ATT&CK et de NVD (National Vulnerability Database).

Sources et références : MITRE ATT&CK · CERT-FR

Questions frequentes

Comment mettre en place Reverse Engineering .NET dans un environnement de production ?

La mise en place de Reverse Engineering .NET en production necessite une planification rigoureuse, incluant l'evaluation des prerequis techniques, la definition d'une architecture cible, des tests de validation approfondis et un plan de deploiement progressif avec des points de controle a chaque etape.

Pourquoi Reverse Engineering .NET est-il essentiel pour la securite des systemes d'information ?

Reverse Engineering .NET constitue un element fondamental de la securite des systemes d'information car il permet de reduire significativement la surface d'attaque, d'ameliorer la detection des menaces et de renforcer la posture globale de securite de l'organisation face aux cybermenaces actuelles.

Faut-il des connaissances en assembleur pour pratiquer Reverse Engineering .NET : Décompilation, Analyse ?

Des bases en x86/x64 sont nécessaires pour le reverse natif. Pour le .NET ou Java, la décompilation produit du code lisible et l'assembleur est moins critique. Commencez par le langage que vous maîtrisez.

Articles connexes

Rétro-Ingénierie
Anti-Rétro-Ingénierie : Techniques des APT
Packing, anti-debug, VM detection, code virtualization
Rétro-Ingénierie
Déobfuscation de Malwares Polymorphes
Techniques de déobfuscation et unpacking avancé
Rétro-Ingénierie & IA
IA et Frameworks d'Analyse de Malwares
Machine learning pour la classification et l'analyse automatisée
Techniques Hacking
Guide Reverse Engineering & Analyse Malware
Méthodologie complète d'analyse de binaires natifs et managés
Threat Intelligence
Infostealers : la Menace Silencieuse du Cybercrime
RedLine, Raccoon, AgentTesla -- écosystème et détection
Techniques Hacking
Évasion EDR/XDR : Techniques et Contre-Mesures
Contournement des solutions de détection endpoint
Techniques Hacking
C2 Frameworks : Mythic, Havoc, Sliver
Architecture, détection et hunting des frameworks C2
Exploitation
Buffer Overflow et Corruption Mémoire
Stack overflow, heap exploitation, ROP chains

Références et ressources externes

- dnSpyEx sur GitHub -- Fork maintenu du décompilateur/debugger .NET
- ILSpy sur GitHub -- Décompilateur .NET open-source
- de4dot sur GitHub -- Déobfuscateur .NET automatique

• Microsoft Learn -- .NET Metadata -- Documentation officielle des métadonnées .NET
• ECMA-335 -- CLI Standard -- Spécification du Common Language Infrastructure

Points clés à retenir

- 9. Protection du code .NET
- 10. Checklist d'analyse .NET
- Questions frequentes

Ayi NEDJIMI Consultants — Expert cybersécurité offensive & intelligence artificielle

ayinedjimi-consultants.fr · ayi@ayinedjimi-consultants.fr

© 2026 — Reproduction interdite sans autorisation.